kdrive 14.1.0

User Guide

kdrive Component Library

KDRIVE

Weinzierl Engineering GmbH

kdrive 14.1.0 User Guide
kdrive Component Library
Edition 1

Author Weinzierl Engineering GmbH *info@weinzierl.de*
kdrive User Guide

1. Introduction **1**
1.1. kdrive Components 2
1.1.1. kdrive KNX 2
1.1.2. kdrive Connector 2
1.1.3. kdrive Access 2
1.1.4. kdrive Kernel 2
1.1.5. kdrive Easy 2
1.1.6. kdrive Services 2
1.1.7. kdrive Object Server 3
1.1.8. kdrive ETS Managed Stack 3
1.2. Requirements 3
1.3. Supported Platforms 3
1.4. kdrive Application Programming Interfaces 4

2. Access **5**
2.1. Access Ports 5
2.2. IP Tunneling 5
2.3. IP Routing 6
2.4. USB 6
2.5. Tiny Serial 6
2.6. FT1.2 Serial 6
2.7. Access Port Enumeration 6
2.8. Creating a Connection 7
2.9. KNX Telegram Formatters 9
2.10. Sending KNX Telegrams 10
2.11. Receiving KNX Telegrams 10
2.12. Local Device Management 11
2.13. Working with Group Value Telegrams 12
2.14. Device Monitor 13

3. Services **14**
3.1. Common Input Parameters 15
3.2. Common Output Parameters 16
3.3. Exceptions 16
3.4. Device Services 17
3.4.1. DeviceDescriptorType0Read 17
3.4.2. DeviceDescriptorType2Read 18
3.4.3. RestartDeviceType0 19
3.4.4. RestartDeviceType1 20
3.4.5. PropertyDescriptionRead 21
3.4.6. PropertyValueRead 23
3.4.7. PropertyValueWrite 24
3.4.8. FunctionPropertyCommand 25
3.4.9. FunctionPropertyStateRead 26
3.4.10. MemoryRead 27
3.4.11. MemoryWrite 29
3.4.12. UserMemoryRead 30
3.4.13. UserMemoryWrite 31
3.4.14. AuthorizeRequest 33
3.4.15. KeyWrite 33
3.4.16. NetworkParameterReadDevice 34
3.4.17. NetworkParameterWriteDevice 36
3.4.18. LoadStateRead 36
3.4.19. LoadStateWrite 37
3.4.20. SwitchProgMode 39

3.4.21. ReadProgMode 41
3.4.22. MemoryBlockRead 42
3.4.23. MemoryBlockWrite 44
3.4.24. ReadDownloadCounter 46
3.4.25. WriteObjectLink 47
3.4.26. ReadObjectLink 48
3.4.27. ObjectIndexRead 49
3.4.28. SendLocalisationCommandDevice 50
3.4.29. FindUnconfiguredDevices 51
3.5. Broadcast Services 53
3.5.1. IndividualAddressProgModeRead 53
3.5.2. IndividualAddressProgModeReadSingle 54
3.5.3. IndividualAddressProgModeWrite 55
3.5.4. IndividualAddressSerialNumberRead 57
3.5.5. IndividualAddressSerialNumberWrite 58
3.5.6. NetworkParameterWriteBroadcast 59
3.5.7. WriteLocalisationMode 60
3.5.8. DomainAddressProgModeRead 60
3.5.9. DomainAddressProgModeWrite 61
3.5.10. DomainAddressSelectiveRead 63
3.5.11. DomainAddressSelectiveType1Read 65
3.5.12. DomainAddressSerialNumberRead 66
3.5.13. DomainAddressSerialNumberWrite 67
3.5.14. FindJustPoweredDevices 68
3.6. Group Services 68
3.6.1. GroupValueWrite 69
3.6.2. GroupValueRead 70
3.7. Reports 71
3.7.1. LocalisationReport 71

4. Object Server **72**
4.1. Object Store 72
4.2. Group Object Associations 72
4.3. Datapoints: Communication Objects 73
4.4. Group Address 74
4.5. Configuration 74
4.6. Event Notifications 74
4.7. ComObject Data 75
4.8. Object Server Usage 75
4.9. Object Server Finite State Machines 76

5. Kernel **83**
5.1. Kernel Manager 83
5.2. Exceptions 83

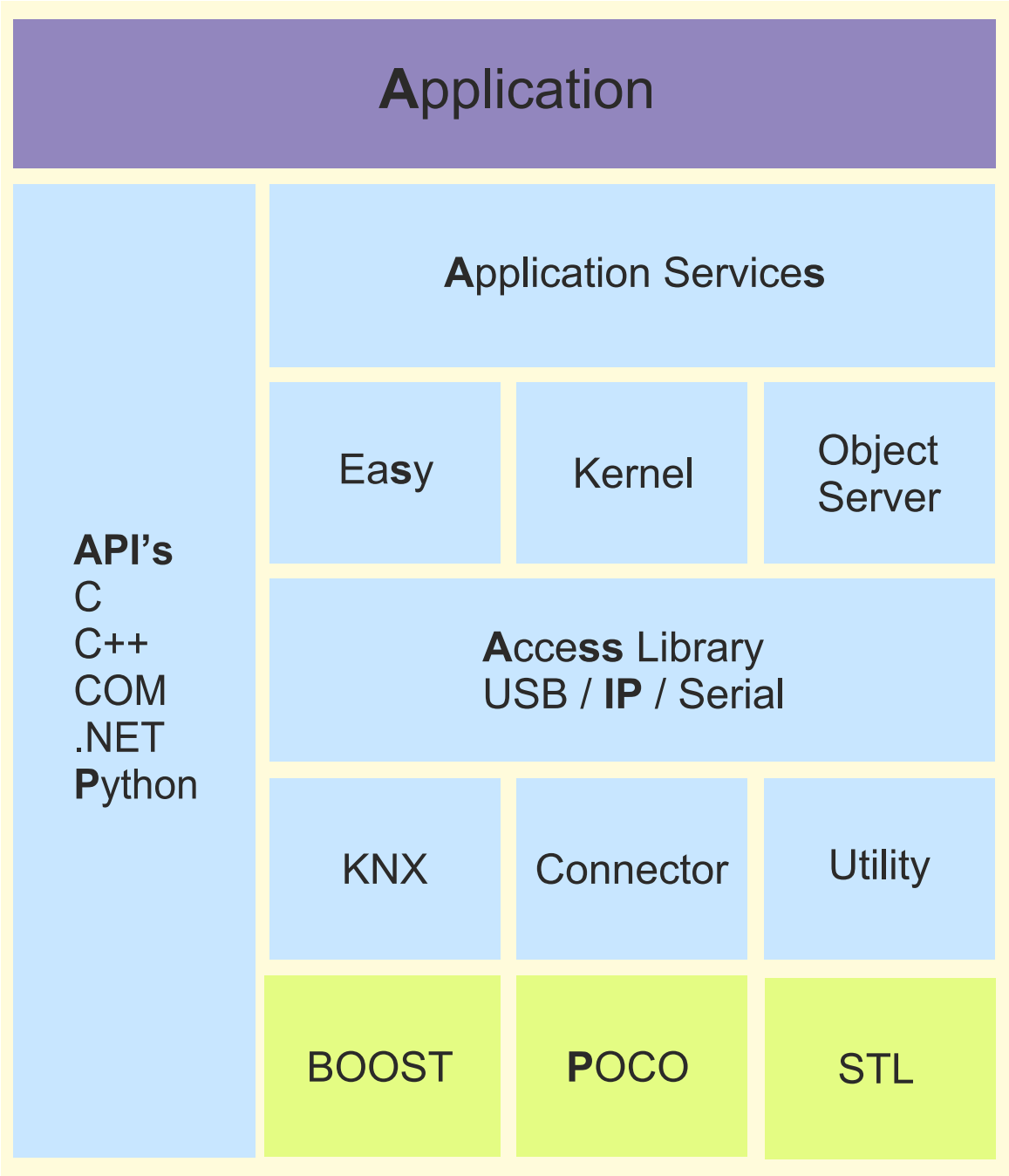
6. kdriveExpress **85**
6.1. Getting Started with kdriveExpress C 85
6.1.1. Error Handling 85
6.1.2. Notification Events 85
6.1.3. Logger 85
6.1.4. Access 86
6.1.5. Services 91
6.1.6. Easy Push Button Services 92
6.1.7. Managed C++ Wrapper 93

7. kdriveExpressCom **94**

7.1. Getting Started with kdriveExpressCom 94
7.1.1. DLL Registration 94
7.1.2. VB6 Add Reference to Project 94
7.1.3. Error Handling 95
7.1.4. API Convention 95
A. FAQ **96**
A.1. Microsoft Visual C++ Redistributable Package 96
B. Revision History **97**
Index **98**

Introduction

kdrive is a cross-platform C++ software component library from Weinzierl Engineering GmbH. It defines a complete ecosystem for the development of KNX applications on platforms with POSIX (like) operating systems, such as Linux, Windows CE and iOS. It uses the *POCO C++ libraries*[1] as its platform abstraction library and should compile and execute on all the systems supported by POCO (i.e. QNX, VxWorks etc).



[1] http://pocoproject.org

1.1. kdrive Components

The following components are available for KNX development:

1.1.1. kdrive KNX

kdrive KNX mostly implements Telegram Formatters (for encoding and decoding KNX Telegrams). The kdrive libraries primarily use the common EMI format (i.e. as the telegram interface to the application etc) and provides conversion routines to and from EM1 and EMI2. Also included are classes for working with Serial Numbers and Individual Addresses.

1.1.2. kdrive Connector

kdrive Connector provides an abstraction layer for sending and receiving "packets", where a packet can be converted to a byte stream. Its classes provide support for receive and transmit threads and asynchronous packet notifications. The kdrive Connector library is used by the kdrive Access library to send and receive KNX telegrams via various media (such as IP, USB and Serial) and provides a uniform interface for sending and receiving telegrams.

1.1.3. kdrive Access

kdrive Access implements KNX Bus Access via Interface Devices. A KNX Interface device enables access to the KNX Bus via a External Messaging Interface (EMI) protocol (of which there are three variants). Supported are EMI1 and EMIC. kdrive Access implements IP Routing, IP Tunnelling, USB, Serial via FT1.2 and Tiny Serial. Synchronous flow control is available as well as buffered asynchonous packet transmission. Packets notifications are provided by the kdrive Connector library.

1.1.4. kdrive Kernel

kdrive Kernel implements the KNX communication stack protocol layers, from the Link Layer up to the Transport Layer. The Link Layer is mostly provided by kdriveAccess. The application layer by the kdriveServices and kdriveObjectServer components. The kernel is responsible for managing the connection-oriented communication, and supports multiple concurrent connections.

1.1.5. kdrive Easy

kdrive Easy implements the KNX Easy Configuration modes. Currently it supports the Easy Push Button Services. An implementation is available for Easy Controller mode however is not yet part of the kdrive libraries as it is not cross-platform. The kdriveEasy services are integrated with the ObjectServer and supports both uni-directional and bi-directional linking.

1.1.6. kdrive Services

kdrive Services enable the development of management clients, and offer a wide-range of management services for device-oriented, system and sytem broadcast messages. These services include Property Value Read and Write (device-oriented), Individual Address Read and Write via Programming Mode or Serial Number (broadcast) and Domain Address Read and Write via Programming Mode or Serial Number (system broadcast).

1.1.7. kdrive Object Server

kdrive Object Server is a Datapoint Object Server and enables application developers to work at the "Datapoint" level. The Object Server is configured either by a configuration file (specifying the group/communication objects and their send and optional receive addresses) or via API. At runtime the Object Server listens for Group Value telegrams (i.e. GroupValueWrite) and keeps track of the values of the associated datapoints. Applications can control the bus by writing the value of a specific datapoint (which will generate a GroupValueWrite event) and can interogate the Object Server at any stage for the current value of an object. The Object Server is not managed by the ETS application however can be configured from an ETS project using a placeholder or "dummy" device.

1.1.8. kdrive ETS Managed Stack

This component is currently in the planning phase and will use the core of the KNX Stack (System B) from Weinzierl. The link layer will be replaced with a kdriveAccess Access Port. For more information contact Weinzierl Engineering.

1.2. Requirements

kdrive is dependent on the Standard Template Library (STL), on the TR1 library (provided by Boost), on the POCO C++ Libraries and on several Boost C++ components, such as boost::signals2 (header only). Some components make use of additional boost libaries such as the unit test framework (boost unit) and language bindings for python (boost python) however these components are optional and can be omitted from the build. kdrive uses the CMake build system, which can generate Makefiles for various platform IDE's and compilers (most notably, Visual Studio, Unix Makefiles, Eclipse, Xcode and KDevelop). With its heavy emphasis on flexibility achieved through standard modern C++ patterns KNX applications developed with kdrive can be compiled for a a wide range of platforms however this naturally comes at the price of additional code size after pulling in the STL and the various platform abstraction layers. The size of a kdrive application depends on whether the library is statically compiled or compiled as dynamically loadable shared objects (.so or .dll) and what components are used. For smaller systems (such as embedded Linux etc) the default SDK can be somewhat reduced by removing some of the optional functionality, including some of the POCO C++ library.

1.3. Supported Platforms

The following platforms are currently actively supported:

Table 1.1. Supported Platforms

Operating System	Compiler	Remarks
Windows XP, Windows 7	Visual Studio	Visual Studio 9.0 and 10.0 and the Express Editions. Support for NMake is also available under CMake
Windows CE 5.0	Visual Studio	Visual Studio 9.0. Not yet tested under the other Visual Studio versions
Linux	GCC	Versions following 4.2.4 should be fine. Earlier versions (i.e. 3.4.6) are probably also ok but have not been tested
Mac OS X	LLVM, GCC	Xcode support is provided by CMake

Operating System	Compiler	Remarks
iOS (Apple iPhone, iPad etc)	LLVM	Xcode support is provided by CMake
Android	GCC	Not yet supported. This can be done on demand, if you require support for Android please let us know. POCO has already been ported to Android.

1.4. kdrive Application Programming Interfaces

The primary kdrive Application Programming Interface (API) is the C++ class interface offered by each of the kdrive components. In addition however there are a number of additional libraries that offer their own API's based on (often a subset) of the kdrive API. These include the kdriveExpress library (C API), the kdriveExpressCom library (Microsoft Com+ DLL) and a number of language wrappers, such as .NET or Python. The following table shows the differences between the various API's in terms of functionality etc. To get started with kdriveExpress, see here: *Chapter 6, kdriveExpress*. To get started with kdriveExpressCom, see here: *Chapter 7, kdriveExpressCom*.

Table 1.2. kdrive Library Edition Comparison

kdrive Component	Feature	*kdriveExpress (C and Com libraries)*	kdrive C++
kdriveAccess	USB	✔	✔
	IP Tunneling	✔	✔
	IP Routing		✔
	TinySerial	✔	✔
	FT1.2 Serial	✔	✔
	Interface Enumeration		✔
	Multiple Connections	✔	✔
	Weinzierl Interface Devices	✔	✔
	Interface Devices from Other Manufacturers	Commercial Version only	✔
	Local Device Management	Individual Address, RF Domain Address	✔
kdriveServices	Management Services	Commercial Version only exception for PropertyValue and ProgMode serivces	✔
kdriveObjectServer	Object Server		✔
kdriveKNX	Telegram Formatters	GroupValue	✔

Access

The kdriveAccess library implements KNX Bus Access via Interface Devices. An Interface Device is a KNX Device that enables, for example, the ETS to access the KNX Bus. Typically the Interface Device connects to Twisted Pair however interface devices also exist for Powerline and RF. For IP you can use the KNXNet/IP routing protocol directly. KNX Interface Devices communicate via the External Messaging Interface (EMI) protocol. There are three variants of the EMI protocol, EMI1, EMI2 and Common EMI (EMIC). kdriveAccess supports both EMI1 and EMIC, while EMI2 is not supported. The kdriveAccess interface is via EMIC. That is, kdriveAccess expects all telegrams in EMIC regardless of whether the interface implements EMI1 or EMIC. If EMI1 the telegrams are automatically converted in the library to/from EMIC. This means that you can write client applications using the EMIC format for both EMI1 and EMIC interface devices. The following list is an overview of where you are likely to find the various protocols:

Table 2.1. EMI Protocol Support

EMI1	EMI2	EMIC
USB Interfaces	Some USB Interfaces and Serial (via FT1.2) Interfaces	IP and newer USB or Serial Interfaces

2.1. Access Ports

An Access Port is a C++ class interface to an Interface Device. Each separate media type has its own Access Port, such as the USB Access Port and the IP Tunnelling Access Port. All ports have a common interface for open, close, read, write etc and a property system (of key, value pairs) which includes both standard and port specific properties.

The Access Port Property System is based on a Dictionary like system that supports key,value pairs. The key is a string and the value is a Poco::DynamicAny, which can be a fundamental type, such as string, int etc as well as a vector. Poco::DynamicAny has an interface for inspecting the type, as well as a number of conversion routines for converting between int and string etc. The Property System is mainly used internally by the Access Ports to hold information such as the arguments for open.

Access Ports also have a notification system for Telegrams and Events. The notification system is based on the Connector class (from kdriveConnector) and implements a separate notification thread (so the internal rx mechanism does not block while the notification is processed by the client application). This means that all telegram and event notifications are not in the context of the main application (i.e. you may need to take care when accessing shared data or writing to a GUI etc).

The following sections presents several examples of working with the Access Port. For more information see the samples and the Doxygen documentation.

2.2. IP Tunneling

IP Tunneling is implemented by the **`TunnelingPort`**. It communicates with an IP Interface Device (for example, the *KNX IP Interface 730*[1] from Weinzierl Engineering). Additionally, all KNX IP Routing devices also implement the KNXNet/IP tunneling protocol. The protocol has two separate connections, the device management connection (for local device management) and the Tunneling connection, for sending and receiving KNX Telegrams. The TunnelingPort implementation opens the management connection as needed, and closes it once the local device management configuration is finished (via timeout). The tunneling connection is kept open via a keep alive mechanism. It is possible (i.e. via

[1] http://weinzierl.de/en/products/730.html

a intermittent WLAN connection) that the logical tunneling connection is disconnected by the device, in which case the Terminated event signal will be sent via the connector/access port EventSignal (see the base Connector class). The TunnelingPort does not handle auto-reconnect and it is up to the application to handle this. The Terminated event happens within the thread context of the notification system and you can assume that the Tunneling Port has been safely closed. That is, it is possible to re-open the Tunneling Port from within the Event signal context. This applies to all Access Port types.

2.3. IP Routing

IP Routing is implemented by the **`RoutingPort`**. It does not require a physical interface device such as the *KNX IP Router 750*[2] from Weinzierl Engineering. IP Routing connections are used to implement KNX IP Devices that communicate via IP.

2.4. USB

USB is implemented by the **`USBPort`**. It communicates with USB interface devices via a custom USB HID protocol (which encapsulates the KNX Telegrams in 64 byte Reports). Currently only supported under Windows and Linux.

2.5. Tiny Serial

The **`TinySerial`** Access Port implements the Link Layer protocol for the TinySerial interface modules from Weinzierl Engineering.

2.6. FT1.2 Serial

The FT1.2 Serial Access Port **`FT12_SerialPort`** implements the FT1.2 Link Layer protocol for Serial Interface devices.

2.7. Access Port Enumeration

Device enumeration enables application developers to discover which interface devices are available for a specified medium. Enumeration is medium specific and is available for the following Access Ports:

Table 2.2. Interface Device Enumeration

Access Port	Remarks
USB	enumeration via hid driver, and filtered via known Vendor and Product ID's
IP Routing	enumeration via KNXNet/IP Search Request. Returns the list of IP addresses and device names
IP Tunneling	enumeration via KNXNet/IP Search Request. Returns the list of IP addresses and device names
Serial via FT1.2	returns the list of available serial ports, list is currently not filtered for knx devices

[2] http://weinzierl.de/en/products/750.html

Access Port	Remarks
Tiny Serial	returns the list of available serial ports, list is currently not filtered for knx devices

Example 2.1. Sample Enumeration: IP Tunneling

TunnelingPort::Ports is a list (std::vector) of Property Collections, where each Property Collection holds the properties for a single interface device. These properties include the name of the interface, its IP Address, and the IP Address of the local network interface adaptor it was found on (i.e. some machines have multiple network adaptors)

```
TunnelingPort::Ports ports;
tunnelingPort.enumerate(ports);
```

Iterate through the list and write the properties to the debug logger

```
BOOST_FOREACH(const PropertyCollection& propertyCollection, ports)
{
 poco_information(logger, propertyCollection.toString());
}
```

2.8. Creating a Connection

Each Interface Device type (USB, IP Tunneling etc) has its own Access Port type. These Access Ports are responsible for the communication between the client application and the interface device, and implement a range of protocols such as KNXNet/IP and FT1.2 for example. To create a connection with an interface device it is neccessary to first configure the Access Port with the interface specific connection data. For IP Tunneling this is the IP Address, for USB the hid device index found via enumerate. Once a connection is open the client can perform Local Device Management (i.e. configuration of the interface device) and send/receive KNX telegrams.

Example 2.2. Sample: Create Connection - IP Tunneling

This code creates a IP Tunneling connection with a known device. Set Preferred Settings configures the local interface device to use standard default settings (that are supported by the interface).

```
TunnelingPort tunnelingPort;
tunnelingPort.open("192.168.1.34");
tunnelingPort.setPreferredSettings();
```

Example 2.3. Sample: Create Connection - IP Tunneling via Enumeration

This code performs device enumeration, selects the first device, configures the Access Port properties (setProperties) and opens the connection (open).

```
TunnelingPort::Ports ports;
tunnelingPort.enumerate(ports);

TunnelingPort tunnelingPort;
tunnelingPort.setProperties(ports.at(0));
tunnelingPort.open();
tunnelingPort.setPreferredSettings();
```

Example 2.4. Sample: Create USB Connection

USB connections have to be opened via enumeration (to determine the HID device path). The code is similar to the IP Tunneling sample.

```
KnxPort::Ports ports;
UsbPortEnumerator::enumerate(ports);

UsbPort usbPort;
usbPort.setProperties(ports.at(0));
usbPort.open();
```

Once enumeration is performed it is cached and can be used by the USB Access Port to access the port properties. So the following shortcut is also possible

```
KnxPort::Ports ports;
UsbPortEnumerator::enumerate(ports);

UsbPort usbPort;
usbPort.open(0);
```

Example 2.5. Sample: Generic Connection Management via Enumeration

The KNX Access Ports are derived from a KnxPort class and can be handled generically via standard C++ polymorphism, such as:

```
KnxPort::Ports ports;
UsbPortEnumerator::enumerate(ports);
const PropertyCollection& p = ports.at(0);
KnxPort::ptr knxPort(new USBPort);

void open(KnxPort::Ptr knxPort, const PropertyCollection& p)
{
 knxPort->setProperties(p);
 knxPort->open();
}
```

2.9. KNX Telegram Formatters

kdriveAccess internally uses the kdriveKnx library to manipulate KNX telegrams. kdriveKnx contains a comprehensive set of telegram formatters which can also be used in application development. A Telegram Formatter implements the rules for building and parsing specific KNX telegram types. We have formatters for L_DATA telegrams (L_DATA_Req, L_DATA_Ind etc) and more complex application layer telegrams such as PropertyValueRead, GroupValueWrite etc. Using the formatters also makes it possible to match incoming telegrams with a specific telegram type. This enables us to easily build complex functionality such as telegram filters etc using a generic framework (C++ templates).

Example 2.6. Sample: Creating a PropertyValueRead Telegram

The following sample creates a PropertyValueRead to read the serial number property. Note, the formatter is responsible for generating the telegram and all access to the telegram attributes is by standard setter and getter functions.

```
A_PropertyValue_Read::Req telegram;
telegram.setDestAddress(0x201);

// set the app data attributes
A_PropertyValue_Read::Req::Data& appData = req.getDataFormatter();
appData.setObjectIndex(0);
appData.setPropertyId(11);
appData.setNumberOfElements(1);
appData.setStartIndex(1);
```

Example 2.7. Sample: Creating a telegram filter for PropertyValueRead

To create a telegram filter we can use the isValid function. We create the formatter and read in a KNX telegram. Calling isValid will indicate whether the telegram type matches the formatter type. So it is possible to filter for all incoming L_DATA_Req telegrams or for a specific telegram type, such as the response to a PropertyValueRead request (see the code below).

Here we get the packet (getPacket mysteriously reads a packet from somewhere and is not part of the library but a placeholder for the sample). All TelegramPackets have a buffer, which we pass to the formatter. The formatter in this case is a PropertyValueRead indication (i.e. the response to our above PropertyValueRead request). If isValid is true it means that the telegram type is correct, and then we can extend our filter with additional information from the getter functions. In this case, the individual address.

```
TelegramPacket::Ptr telegramPacket = getPacket();
TelegramPacket::Buffer& buffer = telegramPacket->getBuffer();

A_PropertyValue_Response::Ind formatter;
formatter.read(buffer);
return formatter.isValid() && (formatter.getSrcAddress() == 0x201);
```

2.10. Sending KNX Telegrams

Telegrams can be sent synchronously or asynchronously. The default configuration is synchronous (blocking) transmission. That is, the access port sends the telegram request to the interface and waits for the L_DATA confirm.

Example 2.8. Sample: Sending a telegram (synchronous)

You can use the kdriveKnx formatters to generate the telegram or simply build it from raw bytes, as in the following example. This sample assumes an open Access Port.

```
const std::vector telegram =
 boost::assign::list_of (0x11)(0x0A)(0x02)(0x08)(0x02)(0x00)(0xC5)(0x00)(0x00)
 (0x00)(0x02)(0x00)(0x2C)(0xE0)(0x00)(0x00)(0x00)(0x01)(0x01)(0x00)(0x81);

TelegramPacket::Ptr telegramPacket(new TelegramPacket);
telegramPacket->setBuffer(telegram);

knxPort->send(telegramPacket);
```

Example 2.9. Sample: Sending a telegram (asynchronous)

To send in asynchronous mode simply enable the tx thread. This sample generates a burst of telegrams which are added to the Tx queue. The telegrams are sent synchronously from the dedicated tx thread, and the application doesn't have to wait while the burst generation is sent.

```
knxPort->startTxThread();

const std::vector telegram =
 boost::assign::list_of (0x11)(0x0A)(0x02)(0x08)(0x02)(0x00)(0xC5)(0x00)(0x00)
 (0x00)(0x02)(0x00)(0x2C)(0xE0)(0x00)(0x00)(0x00)(0x01)(0x01)(0x00)(0x81);

TelegramPacket::Ptr telegramPacket(new TelegramPacket);
telegramPacket->setBuffer(telegram);

for (int i = 0; i < 100; ++i)
{
 knxPort->send(telegramPacket);
}
```

2.11. Receiving KNX Telegrams

Access Ports implement a observer notification pattern for receiving Rx and Tx telegram notifications. These notifications are handled in a dedicated notification thread as part of the Access Port and do not influence the operation of the send or receive threads. That is, clients can process a telegram notification within the notification callback function and will only block the other notification clients, not the internal mechanism of the send and receive threads. Multiple listeners are supported and it is possible to obtain Rx only, Tx only or all (Rx + Tx) telegram notification events.

Example 2.10. Sample: Receiving all telegram signals

```
void connect(AccessPort& accessPort)
{
 AccessPort::PacketSignal& s = accessPort.getPacketSignal();
 s.connect(onPacket);
}

void onPacket(AbstractPacket::Ptr packet, int direction)
{
 // do something with the packet
}
```

2.12. Local Device Management

Local Device Management is specified as part of the EMI protocol and defines how the interface device can be configured from a client application. This includes setting the individial address of the interface device, setting the protocol (EMI protocol, if more than 1 is supported), the layer (link layer, bus monitor) and retrieving the mask version. The following standard interface is implemented:

Table 2.3. Local Device Management Properties

Property	Description
Protocol	Interfaces support one or more EMI protocols (typically one). Supported functions: getSupportedProtocols setProtocol getProtocol isProtocolSupported getPreferredProtocol and setPreferredProtocol
Layer	We support Link Layer, BusMonitor and Raw formats. Supported functions: getSupportedLayers setLayer getLayer isLayerSupported isLayerSupported and getPreferredLayer
Mask Version	Supported functions: getMaskVersion getMaskVersionString
Media Type	Supported functions: getMediaTypes getMediaTypesString
Individual Address	Supported functions: setIndividualAddress getIndividualAddress
Powerline Domain Address	Supported functions: setPlDomainAddress getPlDomainAddress
RF Domain Address	Supported functions: setRfDomainAddress getRfDomainAddress
Serial Number	Supported functions: setSerialNumber getSerialNumber

2.13. Working with Group Value Telegrams

There are three Group Value Telegrams: GroupValue_Write, GroupValue_Read and GroupValue_Response. GroupValue_Write telegrams are sent to transfer information (datapoints) between linked bus participants using a common Group Address. A GroupValue_Read telegram is used to request the value associated with the Group Address and the GroupValue_Response is sent as the response to a GroupValue_Read. The kdriveKnx library has three structures for working with group telegrams, and can be found in: **`kdrive/knx/telegrams/application/GroupValue.h`**. These are the A_GroupValue_Write, A_GroupValue_Read and A_GroupValue_Response structures. Each structure has three components, the Request telegram (Req) the Confirm telegram (Con) and the Indication telegram (Ind). The only difference between the telegrams is the message code (for L_Data_Req etc) otherwise their data formats are identical. When sending a request to the network use the Req field. When receiving a telegram from the network use the Ind field. Con (confim) is used internally to wait for the confirm after sending a request (for flow control). To send a GroupValue_Write request you typically set the Group Address (the telegram destination address) and the datapoint value (setData). Data is encoded in the telegram in two possible ways and depends on the data length. If less than or equal to 6 bits it is stored in the APCI field (and is set with **`setDataByte`**). If greater than 6 bits it is stored as a byte (or more) following the APCI and is set with **`setData`**. To abstract the data formatting of Datapoint Values the kdriveKnx library also implements a Datapoint and GroupValue class. You can use the GroupValue class along with your GroupValue_XXX telegram to read and write standard data lengths.

Example 2.11. Sample: Sending a GroupValue_Write (Request) telegram

```
// create the group value write request
// with the group address hex 901
A_GroupValue_Write::Req groupValueWrite;
groupValueWrite.setDestAddress(0x901);

// Format the data as 8 Bit Unsigned
// using the GroupValue abstraction class
// It writes the data to the groupValueWrite formatter
GroupValue groupValue(groupValueWrite);
groupValue.set8BitUnsigned(0x55);

// Create the telegram packet and send it
TelegramPacket::Ptr telegramPacket(new TelegramPacket(groupValueWrite));
accessPort->send(telegramPacket);
```

Example 2.12. Sample: Get data from a GroupValue_Write (Indication) telegram

```
// Waits for the next packet
// i.e. listens to the packet notification system
TelegramPacket::Ptr telegram = getNextPacket();

// Read the telegram using the GroupValue_Write::Ind formatter
A_GroupValue_Write::Ind groupValueWrite;
groupValueWrite.read(telegram->getBuffer());

// We need to ensure that the telegram is really a
// Group Value Indication, note you could add packet
// filtering to getNextPacket to return only
// GroupValue_Write telegrams
```

```
if (groupValueWrite.isValid())
{
 GroupValue groupValue(groupValueWrite);
 unsigned char data = groupValue.get8BitSigned();
 onData(data);
}
```

Note, you will need exception handling here as read will throw an exception if the telegram is too short for example.

Example 2.13. Sample: Implementing GroupValue_Read

```
// The kdriveAccess library implements a GroupValueReader to send
// a GroupValue_Read request and wait for the (first)
// GroupValue_Response indication.

void readGroupValue(KnxPort& knxPort, unsigned int groupAddress)
{
 TelegramPacket::Ptr packet = GroupValueReader::read(knxPort, groupAddress);

 if (packet)
 {
  A_GroupValue_Response::Ind groupValueResponse;
  groupValueResponse.read(packet->getBuffer());
  GroupValue groupValue(groupValueResponse);
  const std::vector<unsigned char> data = groupValue.getData();
  ... do something with the data, i.e. onData(groupAddress, data);
 }
}
```

2.14. Device Monitor

DeviceMonitor is a statistics gathering object and is primarily used when working with KNX-RF. It can provide individual device information such as Telegram Indication Count, Signal Strength, Battery Status and whether the device is Bi- or Uni- Directional. The DeviceMonitor class connects to the Rx packet signal of an Access Port and auto-adds entries for each device as they are received. This means it is not necessary to configure the DeviceMonitor with device entries. It is possible to enumerate over all devices in the collected device list, or connect to a callback signal to get device information as it changes. For non-RF devices, the DeviceMonitor will simply count the number of telegram indications received.

Services

Services are part of the KNX Application Layer and the kdriveServices component implements the Management Client services. The Management Server is implemented by the KNX Devices that are typically configured with the ETS and are certified by the KNX Association. These devices implement a range of services that are used by the ETS for configuration and management purposes (such as PropertValueWrite etc). The ETS therefore is also seen as a Management Client. A simple service is typically a request/response telegram pair, however some services are a little more complex. Occasionally services will call other services, and it is possible to build complex service architectures using this mechanism.

Services are implemented as synchronous tasks and where applicable provide callback notifications. These notifications can be used for service progress information (i.e. time remaining, or percent complete) and service specific data. Each service defines a set of input and output parameters. Parameters common to all services are outlined directly below, and service specific parameters are outlined in their respective service subsection. The list of common input and output parameters should be considered as ancillary or optional.

The kdriveServices library uses the kdriveAccess library for KNX Bus Access and the kdriveKernel library for the communication stack. There are two main components to the services: a service class (which is responsible for sending, receiving and validating KNX Telegrams) and the ServicePort, which holds the Access Port and the kernel instance. To use the services library you first have to open an AccessPort, create the kernel (which is optional if you only require connection-less services) and create a ServicePort (see the example below). Once you have a ServicePort you can create and run the Services. A Service class has a number of functions for starting and cancelling a service, in addition to input and output properties. Internally the Service classes use these properties to manage inputs and outputs (i.e. similar to the AccessPort properties of key and Poco::DynamicAny value) however there are also classes for each service Input and Output set that define setters and getters. This means you can use the standard IDE auto-complete functionality to view the service parameters or alternatively you can describe a service in text form (i.e. XML) to auto-generate and set the service parameters (using the property system).

The following code sample shows how to create and run the IndividualAddressProgModeRead service. It has a single input for wait time, however it also provides a default value which we use (see the detailed services specification below to determine which service inputs are optional). As this is created without the kernel, it can only be run as connection-less (which is the default).

Example 3.1. Sample Service: IndividualAddressProgModeRead

```
// open the access port
KnxPort::Ptr knxPort = openAccessPort();

// create the service port
ServicePort::Ptr servicePort(new ServicePort(knxPort));

// read the individual address via programming mode
IndividualAddressProgModeRead service(servicePort);
service.run();

// write out the service properties (inputs and outputs)
poco_information(logger, service.toString());
```

To access the service outputs you can use the property system or the getter/setter class wrapper (which simply uses the property system but provides you with typed properties). The following code shows how to access the outputs from the IndividualAddressProgModeRead service:

Example 3.2. Sample Service: IndividualAddressProgModeRead Outputs

```
// initialise the outputs getter/setters with the service
IndividualAddressProgModeRead_Out outputs(service);

// get the individual address list
std::vector<unsigned short> v = outputs.getIndividualAddresses();

// write them out to the logging system
BOOST_FOREACH(unsigned short address, v)
{
 poco_formatter_f1(LOGGER(), "Individual Address %d", static_cast<int>(address));
}
```

Some device services require a transport layer connection (that is, they are connection-oriented). To enable the use of the transport layer you have to use the kdriveKernel library. This can simply be passed to the ServicePort as a parameter, such as in the following example.

Example 3.3. Sample Service: PropertyValueRead (connection-oriented)

```
// open the access port
KnxPort::Ptr knxPort = openAccessPort();

// create the kernel
KernelManager& kernelManager = KernelManager::defaultInstance();
kernelManager.createDefaultLayers(knxPort);
kernelManager.start();

// create the service port
ServicePort::Ptr servicePort(new ServicePort(&kernelManager, knxPort));

// read the individual address via programming mode
PropertyValueRead service(servicePort);
service.setConnectionOriented(true);
service.run(0x201, 0, 11, 1, 1);

// write out the service properties (inputs and outputs)
poco_information(logger, service.toString());
```

3.1. Common Input Parameters

The following input parameters are common to all services.

Table 3.1. Common Input Parameters

Parameter Name	Description
HopCount	The HopCount can be individually set for Group Services, Device Services and Broadcast Services. The default value is 6.

3.2. Common Output Parameters

The following output parameters are common to all services.

Table 3.2. Common Output Parameters

Parameter Name	Description
Start Timestamp	Start timestamp indicates when the service was started
End Timestamp	Indicates when the service was completed or canceled
Error Code	Indicates an error code if the service was not successfully completed (i.e. on service error, or on service cancel)

3.3. Exceptions

The kdriveServices component defines a number of Error Codes (see core/ErrorCodes.h) which also have corresponding Poco::Exception derived Exceptions (i.e. the error codes are propogated throughout the services library using Exceptions). Although the services library defines service specific Exceptions, it is possible that other exceptions types, include kdriveAccess and Poco::Exception based Exceptions are raised (typically from within the other components, however Poco::Timeout is also used in the services library to indicate a timeout condition). The kdrive libraries do not handle the exceptions and the caller must catch them. One simple solution is to catch all exceptions using a Poco::Exception and view the error code. Unique error codes are defined for both the services and access libraries. The following exceptions are used within the general services framework, where a service defines a service specific exception it is outlined in the respective service section:

Table 3.3. Service Framework Exceptions

Exception Name	Description
ServicesException	kdriveServices exception. This is the base class for all Service Exceptions and is not directly thrown, however can be used to catch all Service-based exceptions.
UnknownServiceTypeException	Thrown from the ServiceFactory create function when a service type name is not recognised
AccessPortNotSetException	If a ServicePort has a null pointer for its AccessPort it will raise an AccessPortNotSetException when ServicePort::getAccessPort() is called.
KernelManagerNotSetException	If a ServicePort has a null pointer for its KernelManager it will raise an KernelManagerNotSetException when ServicePort::getKernelManager() is called.
ServicePortNotSetException	Thrown when the service attempts to access a null ServicePort pointer (via getServicePort). The ServicePort is set by the user and as long as the ServicePort has been set this Exception will not be thrown.
OperationCancelledException	Called when the service is interrupted via a cancel event. Note: the cancel event is not always checked, that is some services simply send a single telegram. Typically the cancel event is checked while waiting for a response telegram. Important: you will not al-

Exception Name	Description
	ways receive a OperationCancelledException when the service is cancelled. To be certain, you should use Service::isCanceled instead.
NoResponseFromDeviceException	Base exception for both NegativeConfirmException and ResponseTimeoutException. Added due to the differences in handling between RF and Twisted Pair. i.e. the L_DATA_Con for TP can have an error flag to indicate that it is a negative acknowledge. This isn't supported for RF.
NegativeConfirmException	When a L_DATA_Con is received with the Confirm Flag set to Error (i.e. indicates that a Link Layer Acknowledge was not received by the interface) a NegativeConfirmException is thrown. This is not supported for all media (i.e. supported on TP but not RF).
ResponseTimeoutException	Indicates that an expected response was not received from the device.
InvalidInputParameterException	Services check the input parameters (where possible) against a range of values (these values are defined below for each service). When a given input parameter is out of range this Exception is thrown (only during the run function)

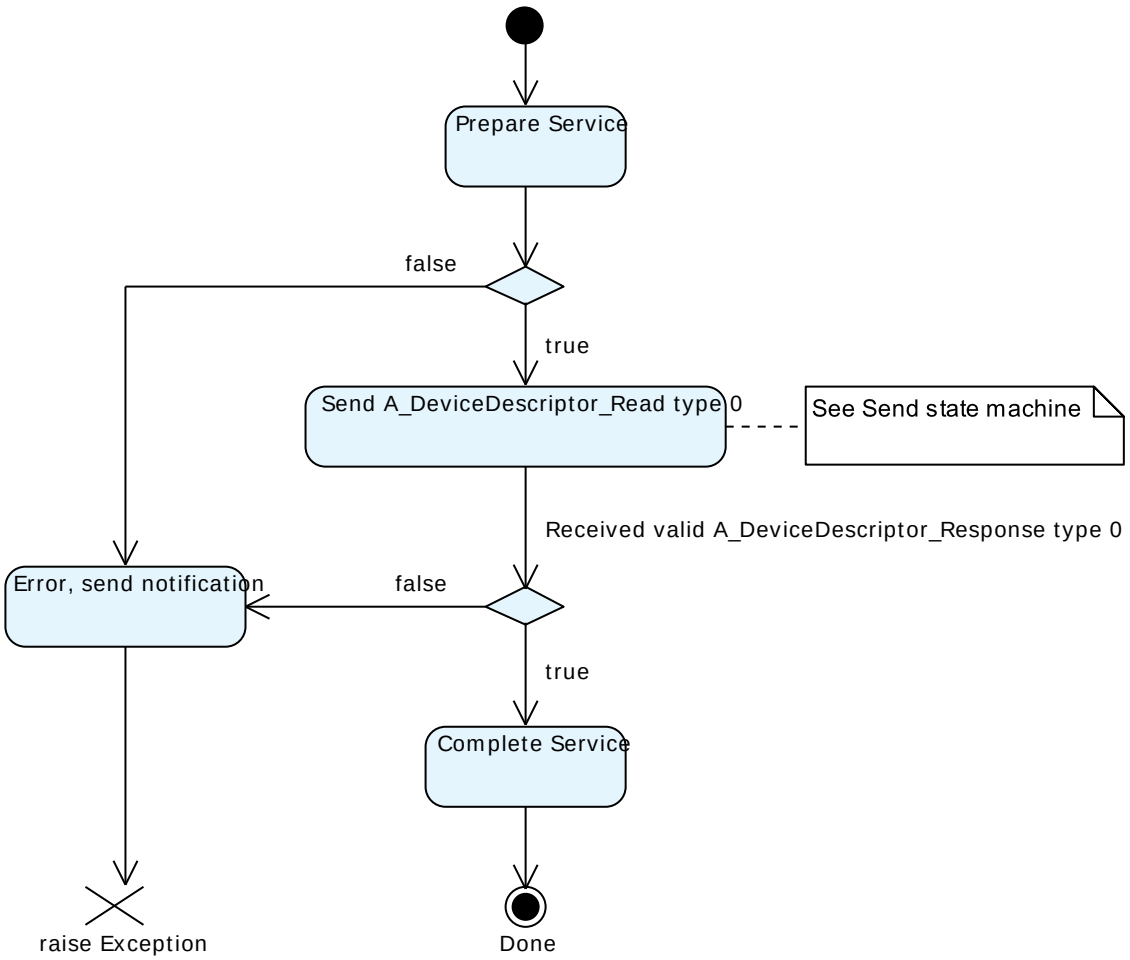
3.4. Device Services

A device service is a service that communicates with a single bus participant (device). Devices are uniquely identified by an Individual Address. For RF devices are identified by an Extended Individual Address which includes the Domain Address of the device. Device services can be sent as either connection-oriented or connection-less. Connection-oriented services require a kdriveKernel instance to manage the communication stack.

3.4.1. DeviceDescriptorType0Read

Table 3.4. DeviceDescriptorType0Read

Attribute	Description
Name	DeviceDescriptorType0Read
Description	Reads Device Descriptor 0 (Mask Version)
Inputs	Individual Address (integer) [1 .. 0xFFFF]
Outputs	Mask Version (integer)
Validation	Via Service Response
Exceptions	NegativeResponseException. Thrown when the device sends a negative response (the type is set to 0x3F). DeviceErrorException. Thrown when the device sends a invalid device descriptor response.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.2.1 A_DeviceDescriptor_Read-service



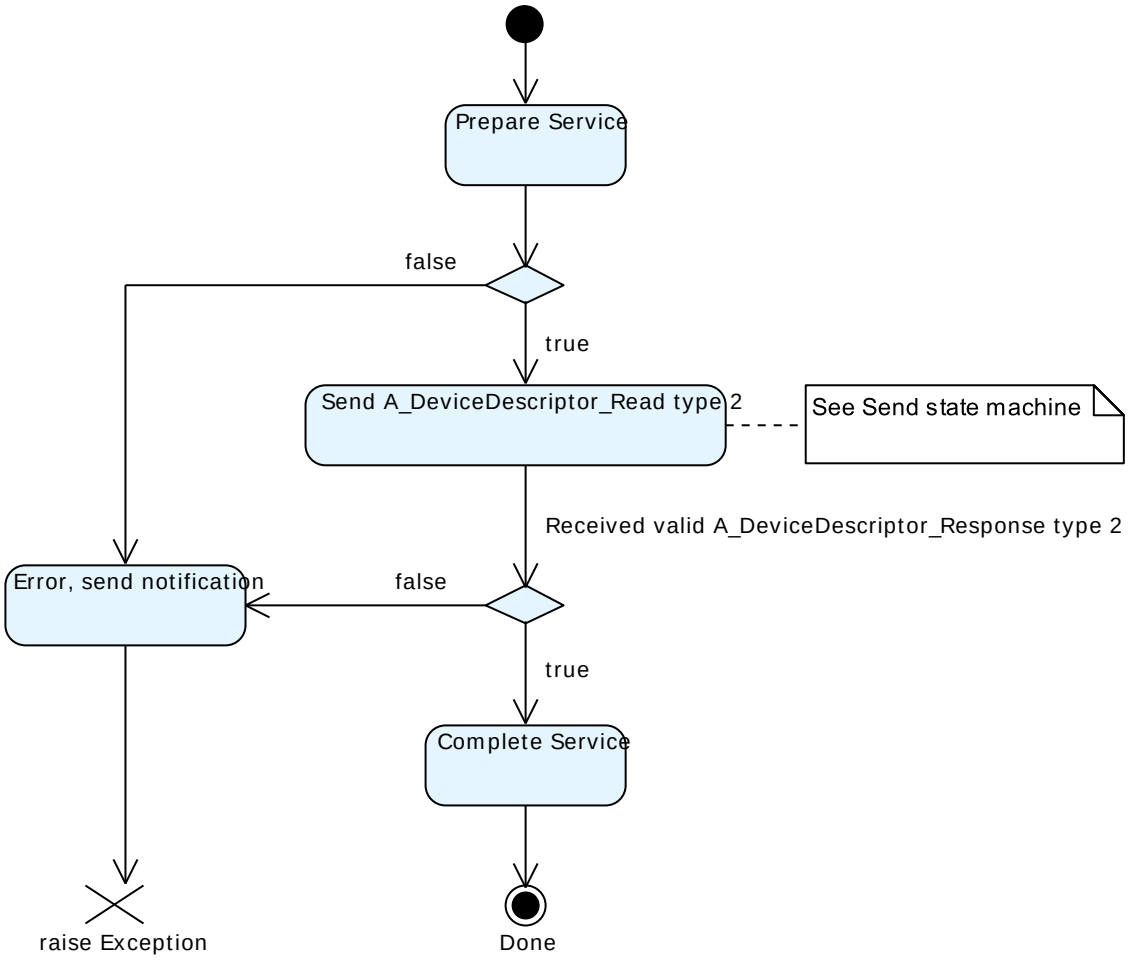
Finite State Machine: DeviceDescriptorType0Read

3.4.2. DeviceDescriptorType2Read

Table 3.5. DeviceDescriptorType2Read

Attribute	Description
Name	DeviceDescriptorType2Read
Description	Reads Device Descriptor 2 (Easy Channel Information). This service sends a DeviceDescriptorType2Read request and waits for the DeviceDescriptorType2Read response. It is possible that the addressed device does not support DeviceDescriptorType2Read and will send an error response instead.
Inputs	Individual Address (integer) [1 .. 0xFFFF]
Outputs	Application Manufacturer Code (integer) Application ID (integer) Application Version (integer) Management Profile (integer) Channel Count Type A (integer) Channel Code Type A (integer) Channel Count Type B (integer) Channel Code Type B (integer) Channel Count Type C (integer) Channel Code Type C (integer)

Attribute	Description
	Channel Count Type D (integer) Channel Code Type D (integer)
Validation	Via Service Response
Exceptions	NegativeResponseException. Thrown when the device sends a negative response (the type is set to 0x3F). DeviceErrorException. Thrown when the device sends a invalid device descriptor response.
Connection Status (default)	Connection-less
Specification	AN134 v02 Flexible E-Mode Channels DV 2.3.1 Device Descriptor Type 2



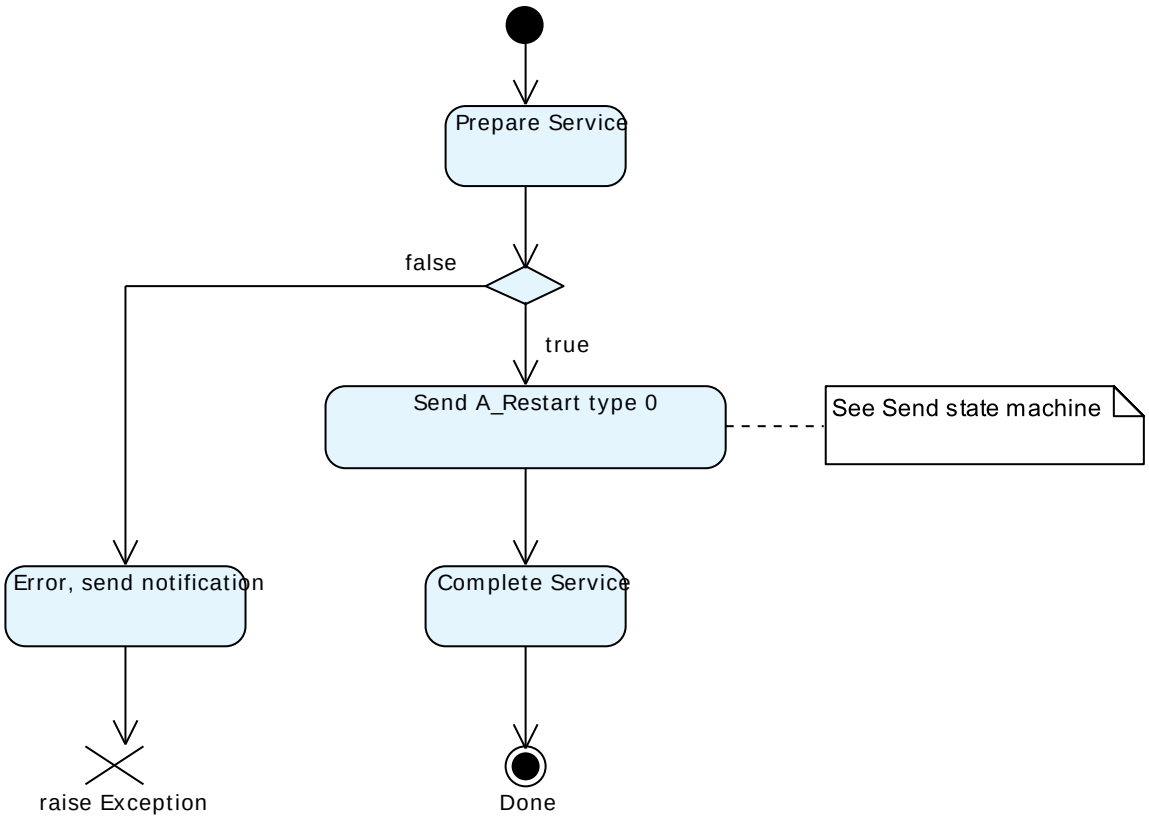
Finite State Machine: DeviceDescriptorType2Read

3.4.3. RestartDeviceType0

Table 3.6. RestartDeviceType0

Attribute	Description
Name	RestartDeviceType0
Description	Restarts the device using Restart (type 0). There is no response to this telegram so it is not possible to know when the device is

Attribute	Description
	ready. For connection-oriented: remote devices do not acknowledge the Restart service. To avoid a T-ACK exception from the transport layer we catch the exception internally in the service and auto-disconnect the connection by sending a Disconnect request.
Inputs	Individual Address (integer) [1 .. 0xFFFF]
Outputs	No outputs
Validation	None
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.2.2 A_Restart-service



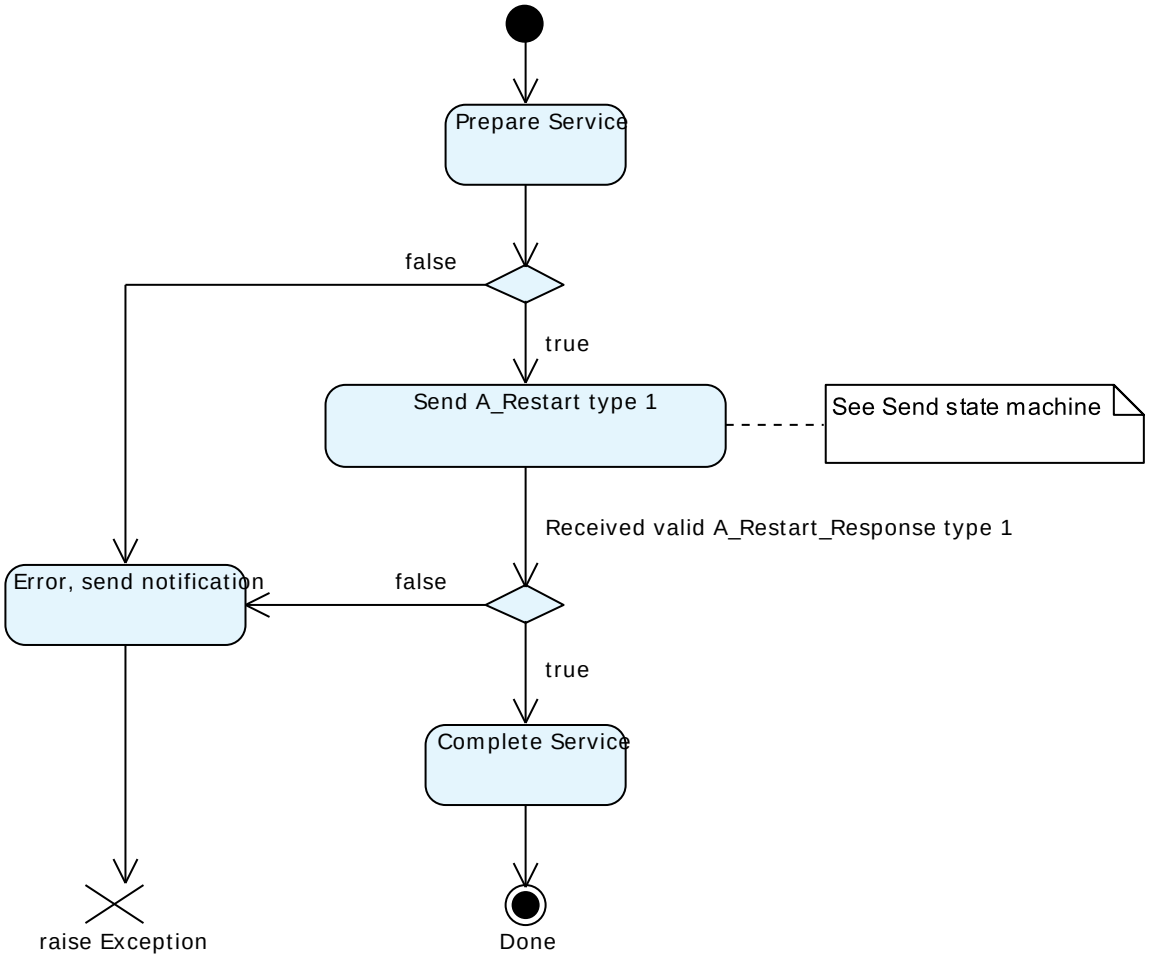
Finite State Machine: RestartDeviceType0

3.4.4. RestartDeviceType1

Table 3.7. RestartDeviceType1

Attribute	Description
Name	RestartDeviceType1
Description	Restarts the device using Restart (type 1)
Inputs	Individual Address (integer) [1 .. 0xFFFF] Erase Code (integer) [0 .. 6] Channel Number (integer) [0 .. 0xFF]
Outputs	Error Code (integer)

Attribute	Description
	Proccess Time (integer)
Validation	Via Service Response
Exceptions	InvalidInputParameterException. Thrown additionally when the erase code is ConfirmedRestart, ResetIA or ResetAP and the channel number is not 0
Connection Status (default)	Connection-less
Specification	AN127 v04 Master Reset DV



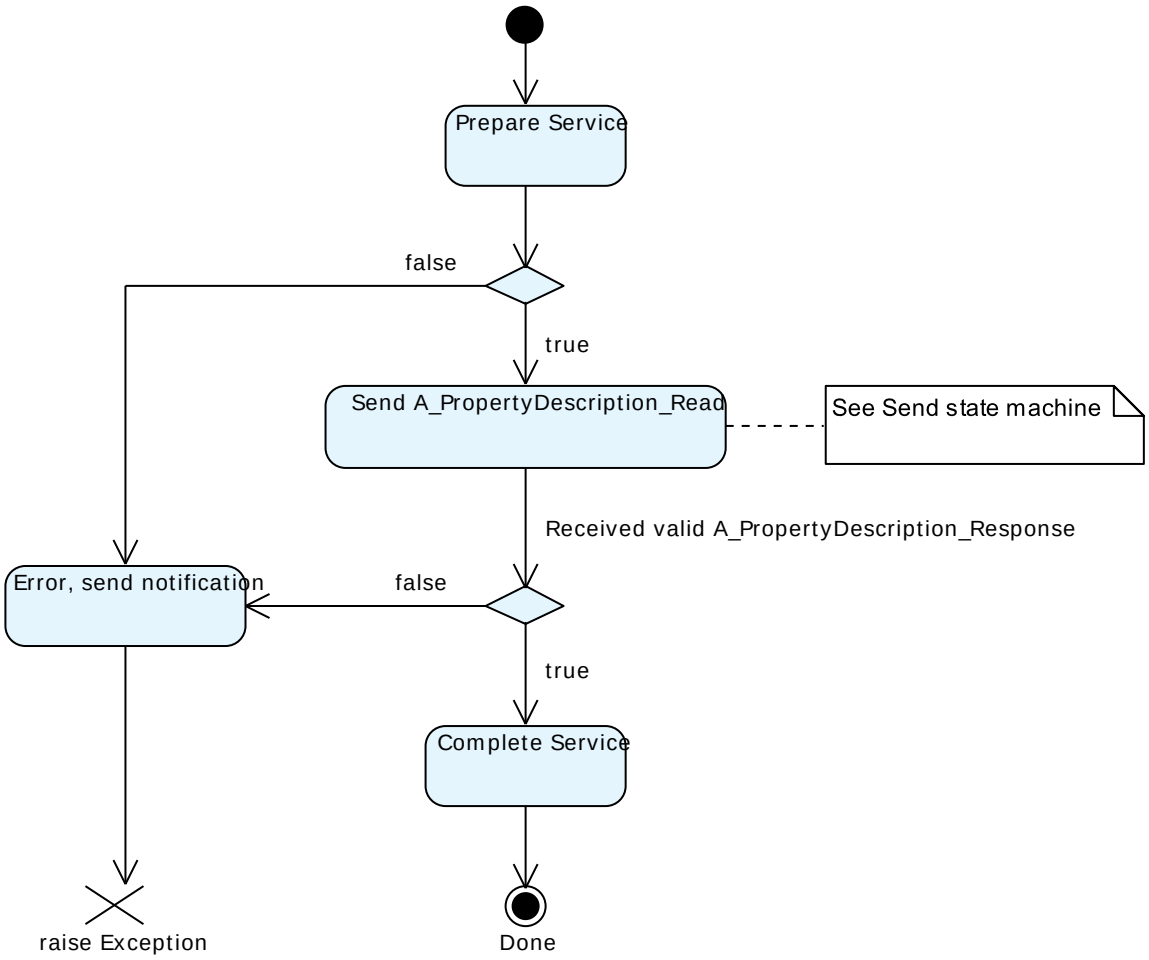
Finite State Machine: RestartDeviceType1

3.4.5. PropertyDescriptionRead

Table 3.8. PropertyDescriptionRead

Attribute	Description
Name	PropertyDescriptionRead
Description	Reads a property description from a device
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [0 .. 0xFF] Property ID (integer) [0 .. 0xFF] Property Index (integer) [0 .. 0xFF]

Attribute	Description
Outputs	Object Index (integer) Property ID (integer) Property Index (integer) Write Enable (boolean) Data Type (integer) Max Number of Elements (integer) Read Access Level (integer) Write Access Level (integer)
Validation	Via Property Response
Exceptions	NegativeResponseException. Thrown when the maximum number of elements in the response is 0. For example: the requested property does not exist. DeviceErrorException. Thrown when either the ObjectIndex, PropertyId or PropertyIndex do not match the associated request parameters. Note: in the case of PropertyId and PropertyIndex only the non-zero input values are checked in the response unless they are both zero, in which case PropertyIndex is checked.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.3.3 A_PropertyDescription_Read-service

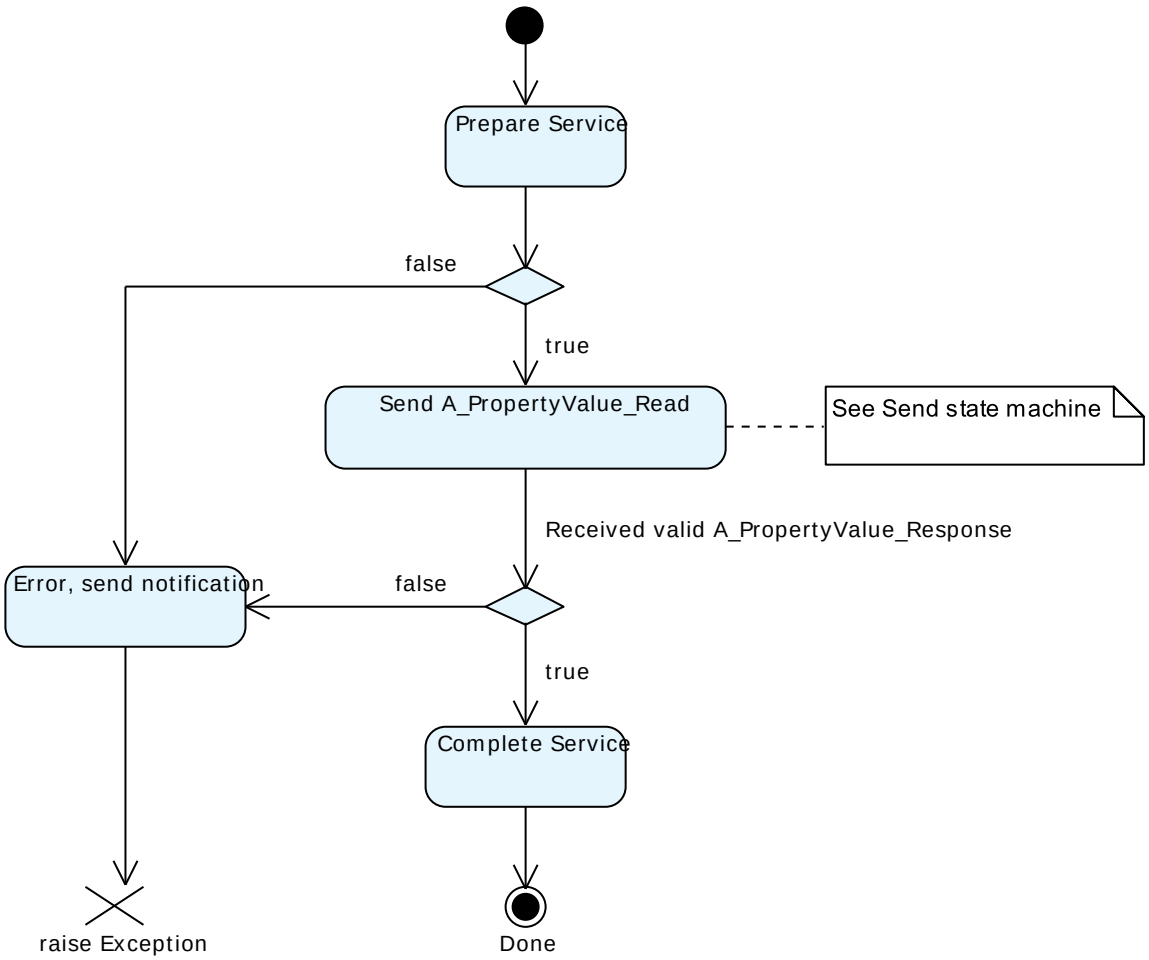


Finite State Machine: PropertyDescriptionRead

3.4.6. PropertyValueRead

Table 3.9. PropertyValueRead

Attribute	Description
Name	PropertyValueRead
Description	Reads a property value from a device
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [0 .. 0xFF] Property ID (integer) [1 .. 0xFF] Element Count (integer) [1 .. 10] Start Index (integer) [0 .. 0xFFF]
Outputs	Object Index (integer) Property ID (integer) Element Count (integer) Start Index (integer) Data (byte array)
Validation	Via Property Response
Exceptions	NegativeResponseException. Thrown when the number of elements in the response is 0. For example: the requested property does not exist, or it is of type PDT_Function etc.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.3.1 A_PropertyValue_Read-service



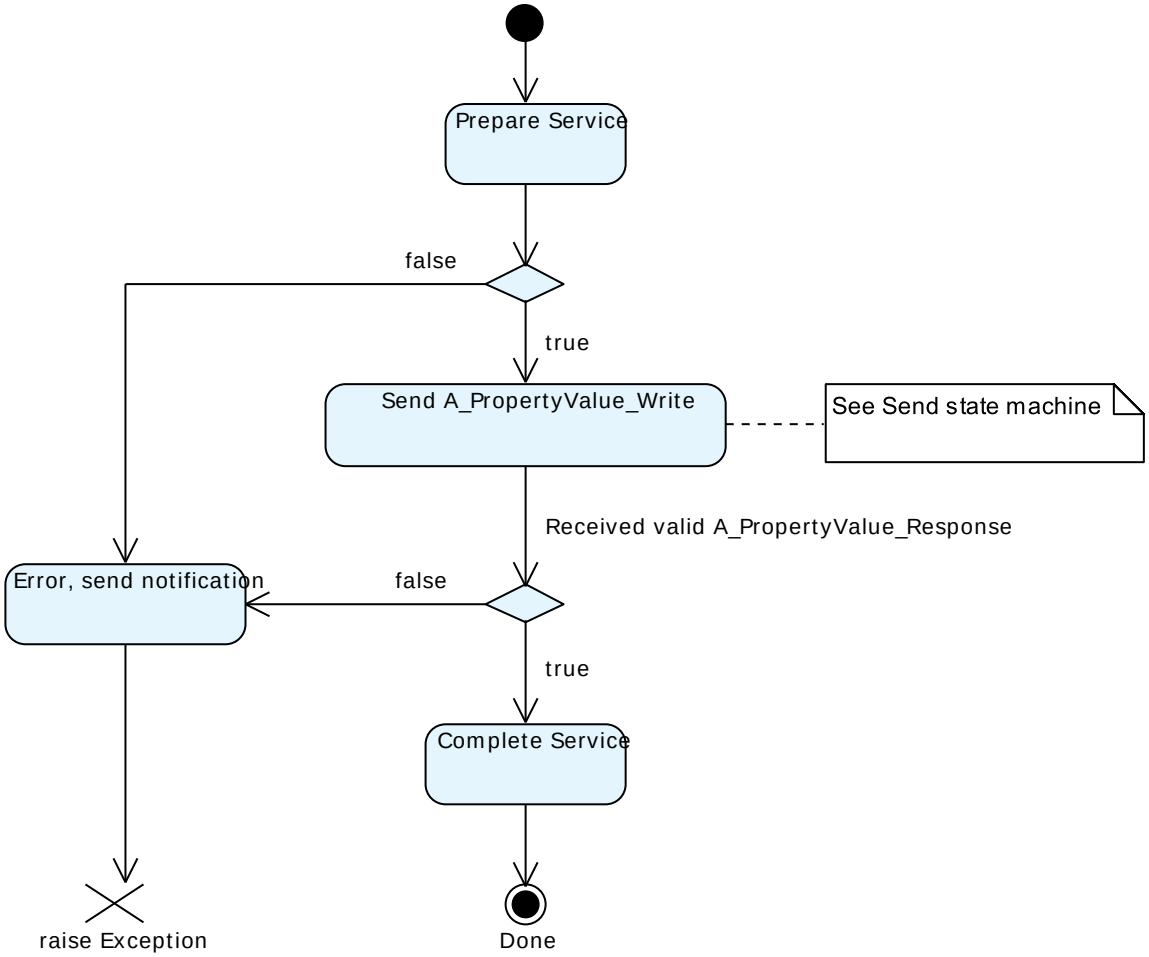
Finite State Machine: PropertyValueRead

3.4.7. PropertyValueWrite

Table 3.10. PropertyValueWrite

Attribute	Description
Name	PropertyValueWrite
Description	Writes the value of a property to a device. If Input Parameter "Verify Response Data" is true then verify the field data in the response with Input Parameter Data.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [0 .. 0xFF] Property ID (integer) [1 .. 0xFF] Element Count (integer) [1 .. 10] Start Index (integer) [0 .. 0xFFF] Data (byte array) [1 .. 10 bytes] Verify Response Data (boolean, optional default = true)
Outputs	Object Index (integer) Property ID (integer) Element Count (integer) Start Index (integer) Data (byte array)
Validation	Via Property Response

Attribute	Description
Exceptions	NegativeResponseException. Thrown when the number of elements in the response is 0. For example: the requested property does not exist, or it is of type PDT_Function, or it is read-only, etc.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.3.2 A_PropertyValue_Write-service



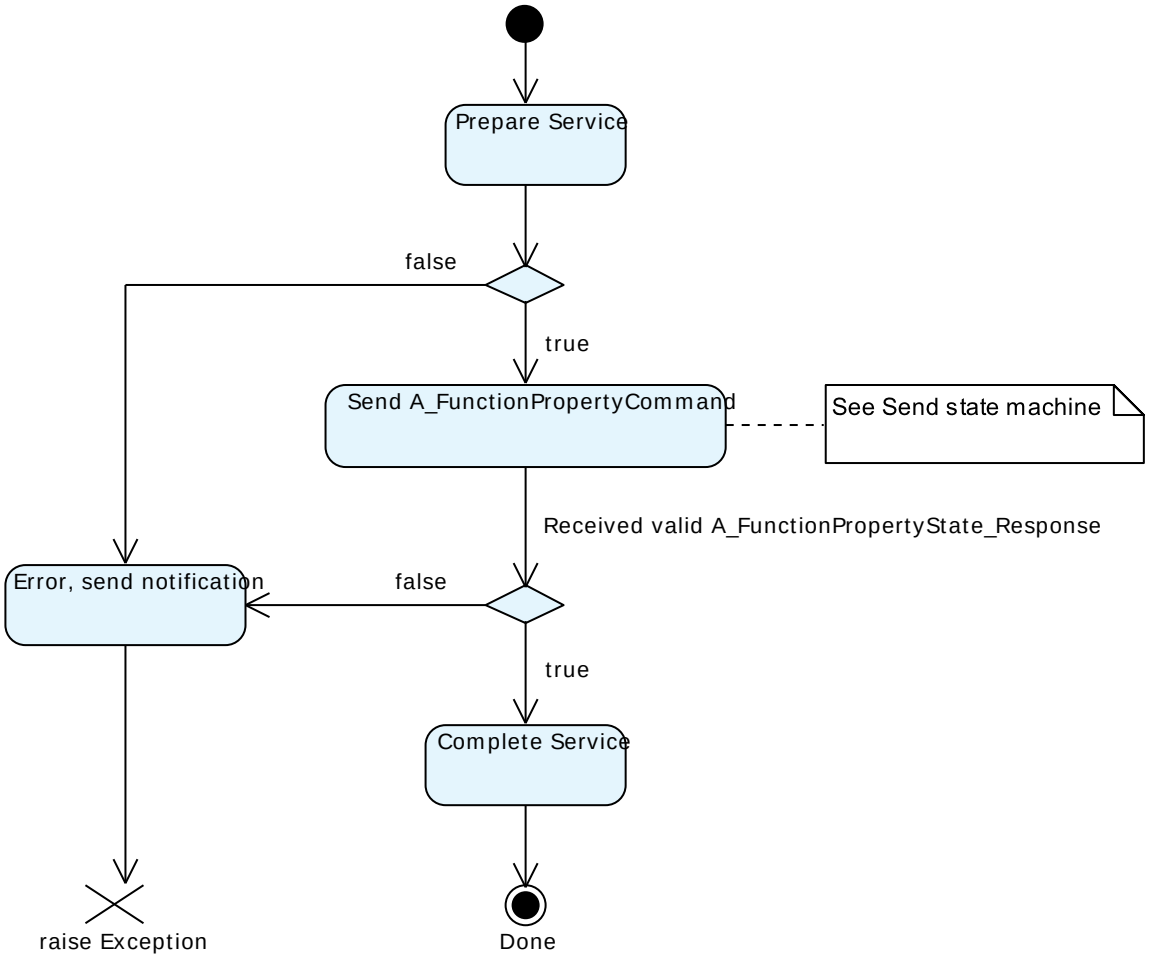
Finite State Machine: PropertyValueWrite

3.4.8. FunctionPropertyCommand

Table 3.11. FunctionPropertyCommand

Attribute	Description
Name	FunctionPropertyCommand
Description	Calls a Function Property of an Interface Object in a device
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [0 .. 0xFF] Property ID (integer) [1 .. 0xFF] Function Specific Input Data (byte array) [1 .. 12 bytes]
Outputs	Object Index (integer) Property ID (integer)

Attribute	Description
	Return Code (integer) Function Specific Output Data (byte array)
Validation	Via Property Response
Exceptions	NegativeResponseException. Thrown when a response is received without return_code and data. For example: the requested property does not exist, or it is not of type PDT_Function.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.5.1 A_FunctionPropertyCommand-Service



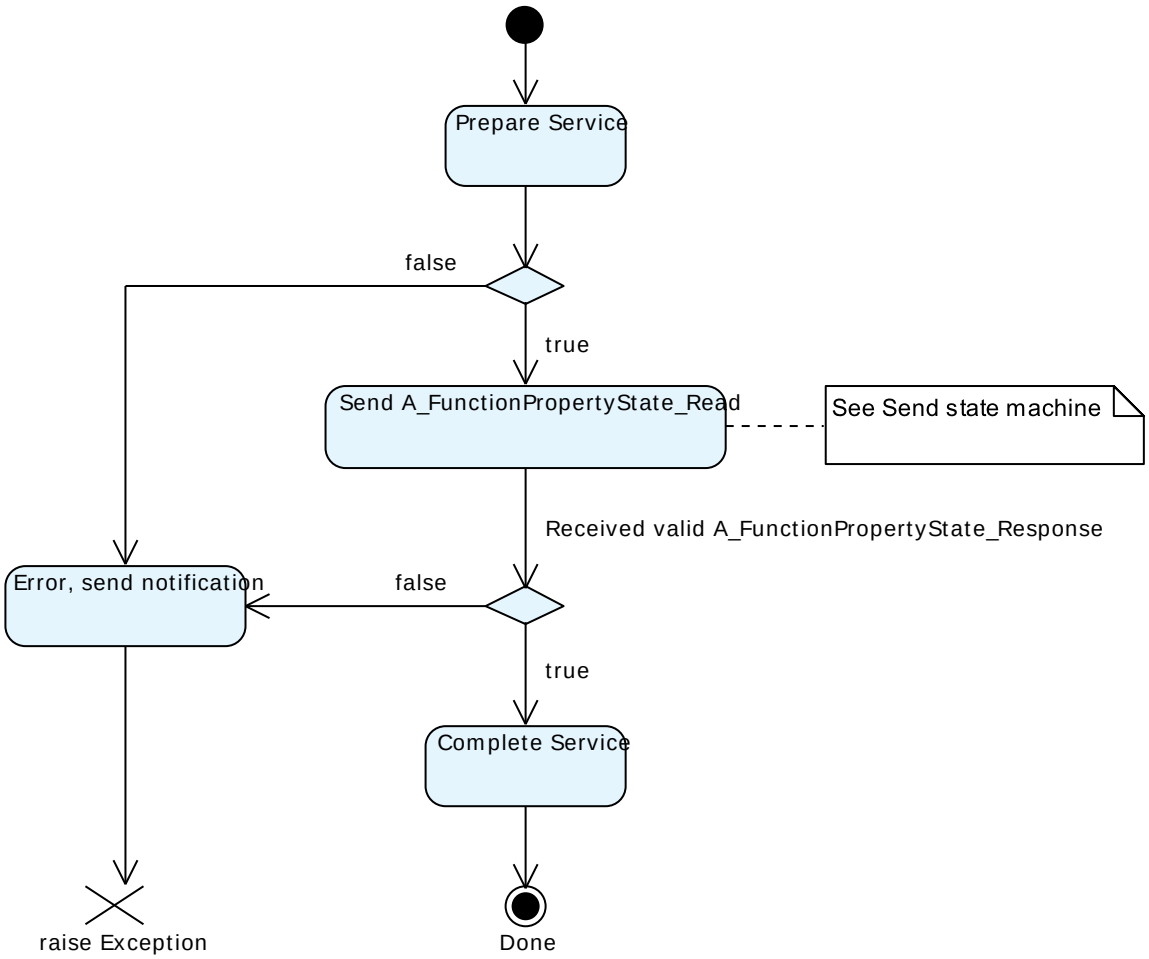
Finite State Machine: FunctionPropertyCommand

3.4.9. FunctionPropertyStateRead

Table 3.12. FunctionPropertyStateRead

Attribute	Description
Name	FunctionPropertyStateRead
Description	Calls a Function Property of an Interface Object in a device
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [0 .. 0xFF]

Attribute	Description
	Property ID (integer) [1 .. 0xFF] Function Specific Input Data (byte array) [1 .. 12 bytes]
Outputs	Object Index (integer) Property ID (integer) Return Code (integer) Function Specific Output Data (byte array)
Validation	Via Property Response
Exceptions	NegativeResponseException. Thrown when a response is received without return_code and data. For example: the requested property does not exist, or it is not of type PDT_Function.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.5.1 A_FunctionPropertyStateRead-Service



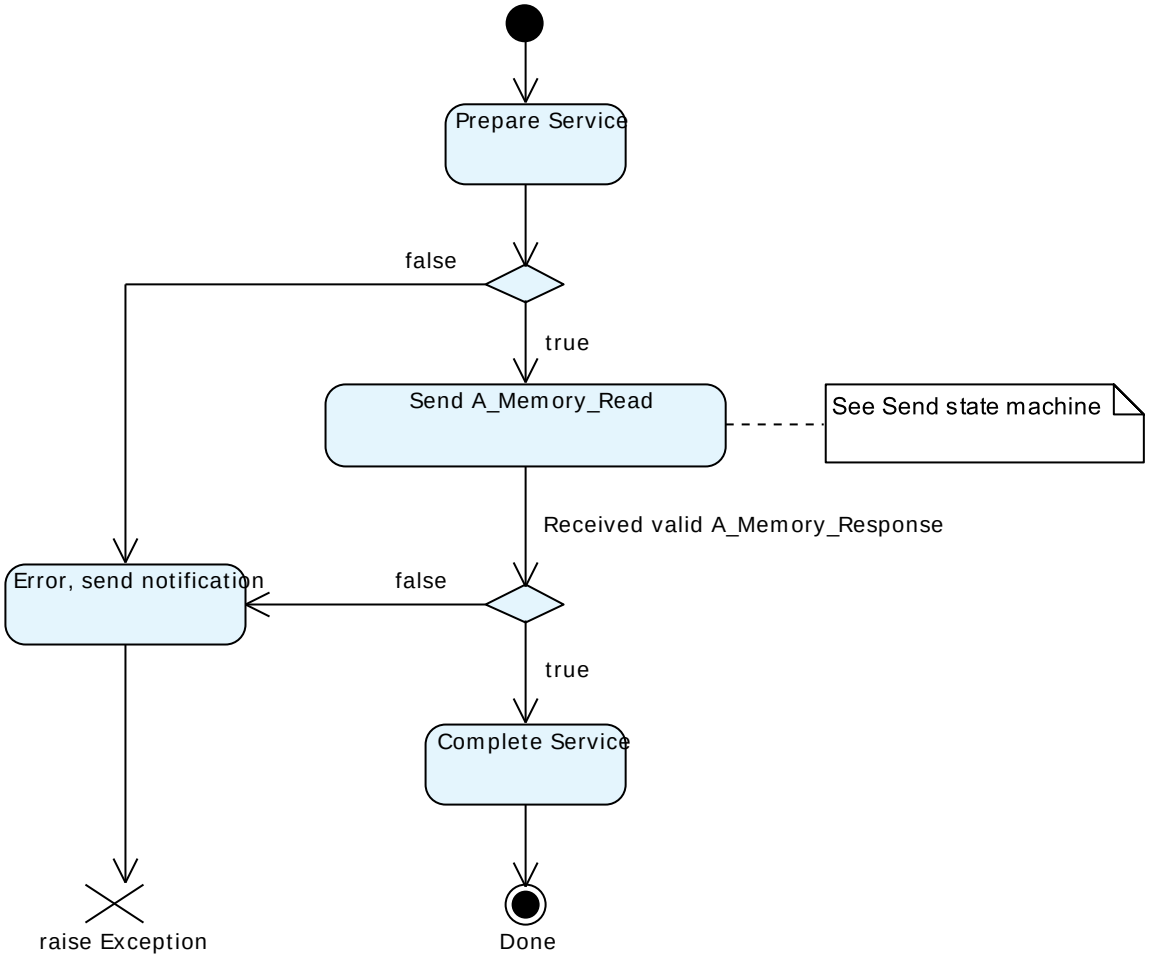
Finite State Machine: FunctionPropertyStateRead

3.4.10. MemoryRead

Table 3.13. MemoryRead

Attribute	Description
Name	MemoryRead

Attribute	Description
Description	Reads data via DMA
Inputs	Individual Address (integer) [1 .. 0xFFFF] Address 16 Bit (integer) [0 .. 0xFFFF] Number (integer) [1 .. 12]
Outputs	Address 16 Bit (integer) Number (integer) Data (byte array)
Validation	Via Memory Response
Exceptions	NegativeResponseException. Thrown when a response is received with Number = 0. For example: The requested address space is unreachable or protected or an illegal number of octets are requested. DeviceErrorException. Thrown when either the Address or the Number in the response do not match those in the request.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.5.3 A_MemoryRead-Service

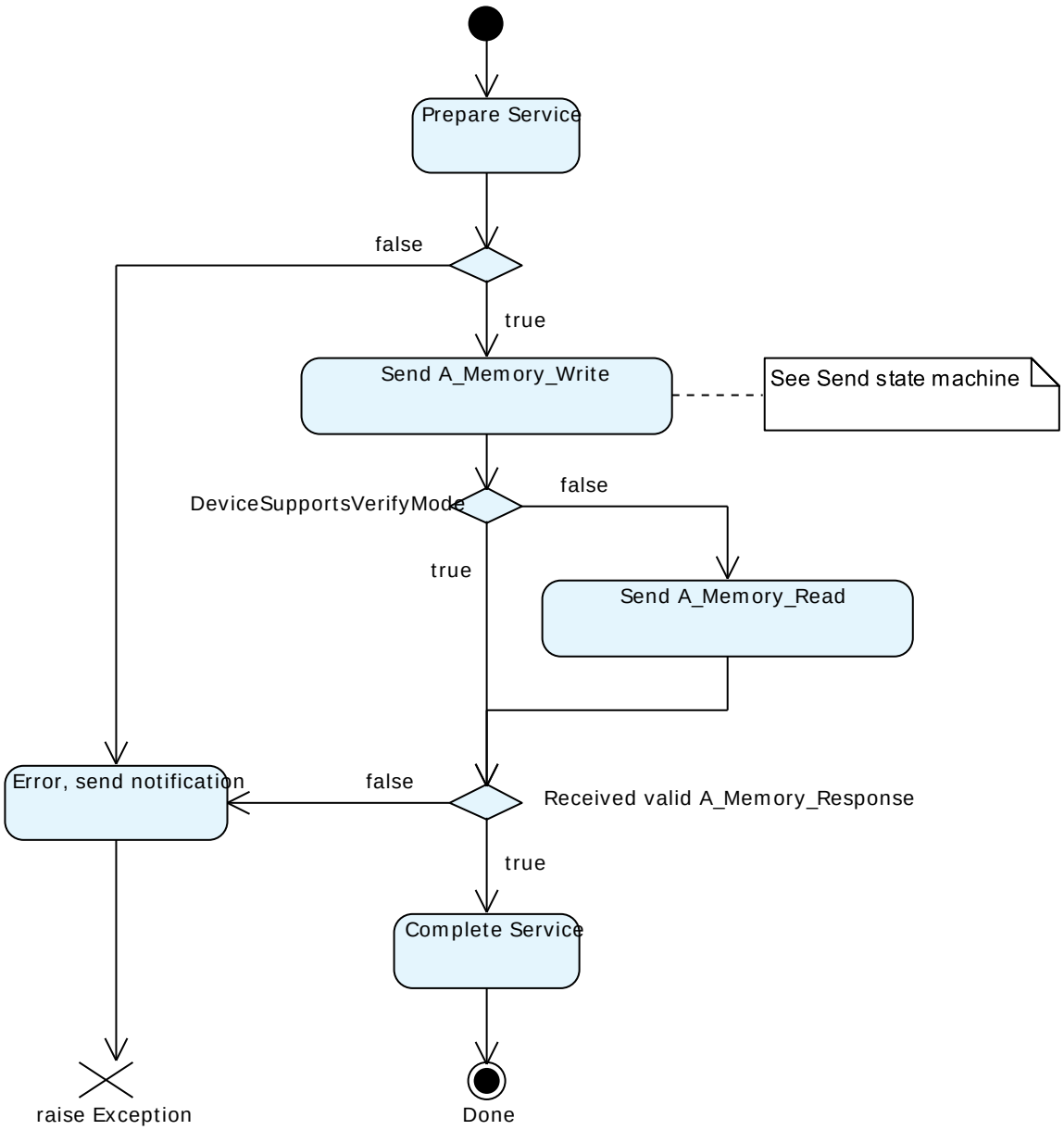


Finite State Machine: MemoryRead

3.4.11. MemoryWrite

Table 3.14. MemoryWrite

Attribute	Description
Name	MemoryWrite
Description	Writes data via DMA. If the device supports the verify mode (= send a A_Memory_Response after rcv a A_Memory_Write), than the service waits for the A_Memory_Response after sends the A_Memory_Write. Otherwise (DeviceSupportsVerifyMode = false) than the service sends a A_Memory_Read and waits for the A_Memory_Response. If Input Parameter "Verify Response Data" is true then verify the field data in the response with Input Parameter Data. Note: The field number is set to the size from input parameter Data.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Address 16 Bit (integer) [0 .. 0xFFFF] Data (byte array) [1 .. 12 bytes] DeviceSupportsVerifyMode (bool, optional default = false) Verify Response Data (boolean, optional default = true)
Outputs	Address 16 Bit (integer) Number (integer) Data (byte array)
Validation	Via Memory Response
Exceptions	NegativeResponseException. Thrown when a response is received with Number = 0. For example: The requested address space is unreachable or protected or an illegal number of octets are requested. DeviceErrorException. Thrown when either the Address, Number or the Data (only if input parameter "Verify Response Data" is true) in the response do not match those in the request.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.5.4 A_Memory_Write-service



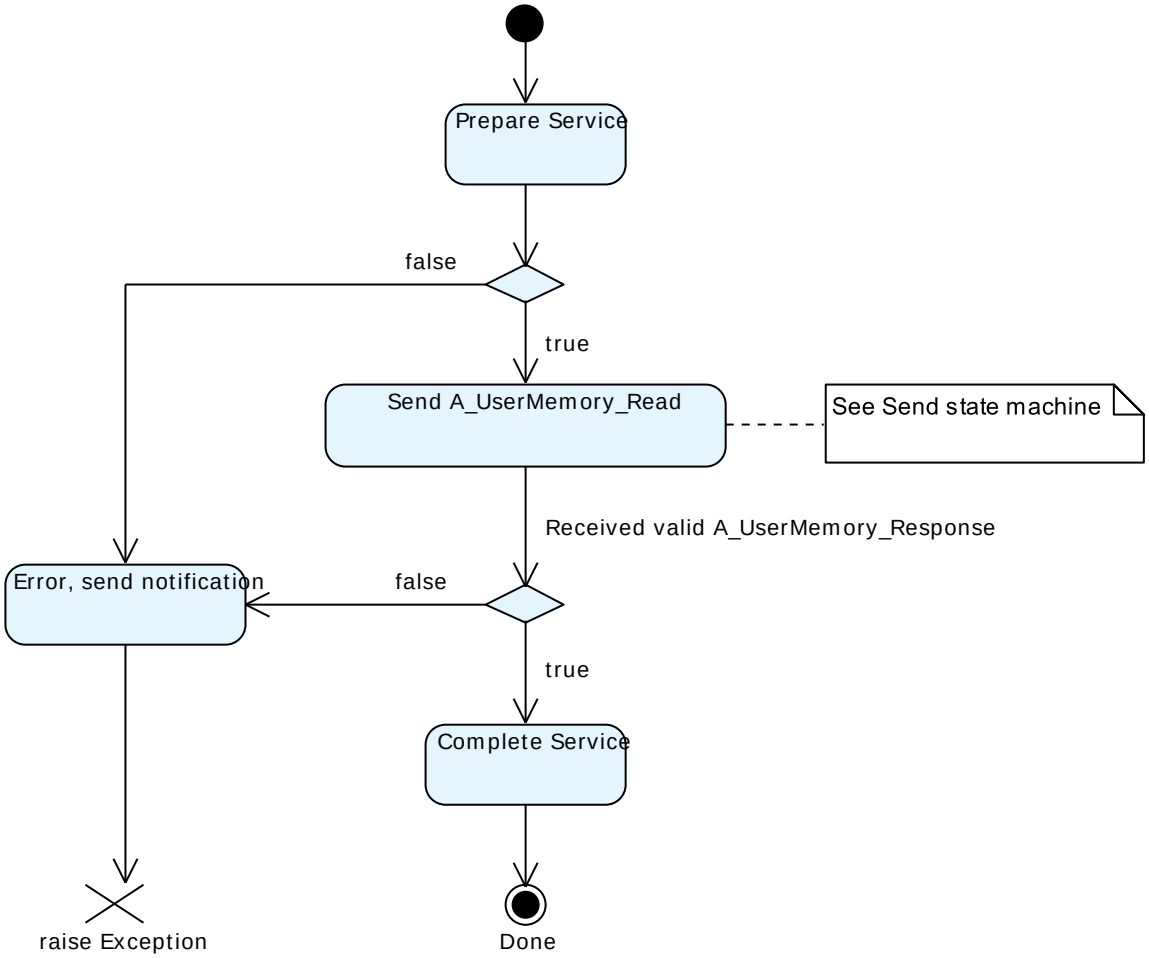
Finite State Machine: MemoryWrite

3.4.12. UserMemoryRead

Table 3.15. UserMemoryRead

Attribute	Description
Name	UserMemoryRead
Description	Reads data via DMA
Inputs	Individual Address (integer) [1 .. 0xFFFF] Address 20 Bit (integer) [0 .. 0xFFFFF] Number (integer) [1 .. 11]
Outputs	Address 20 Bit (integer) Number (integer) Data (byte array)
Validation	Via Memory Response

Attribute	Description
Exceptions	NegativeResponseException. Thrown when a response is received with Number = 0. For example: The requested address space is unreachable or protected or an illegal number of octets are requested. DeviceErrorException. Thrown when either the Address or the Number in the response do not match those in the request.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.4.5.1 A_UserMemoryRead-Service



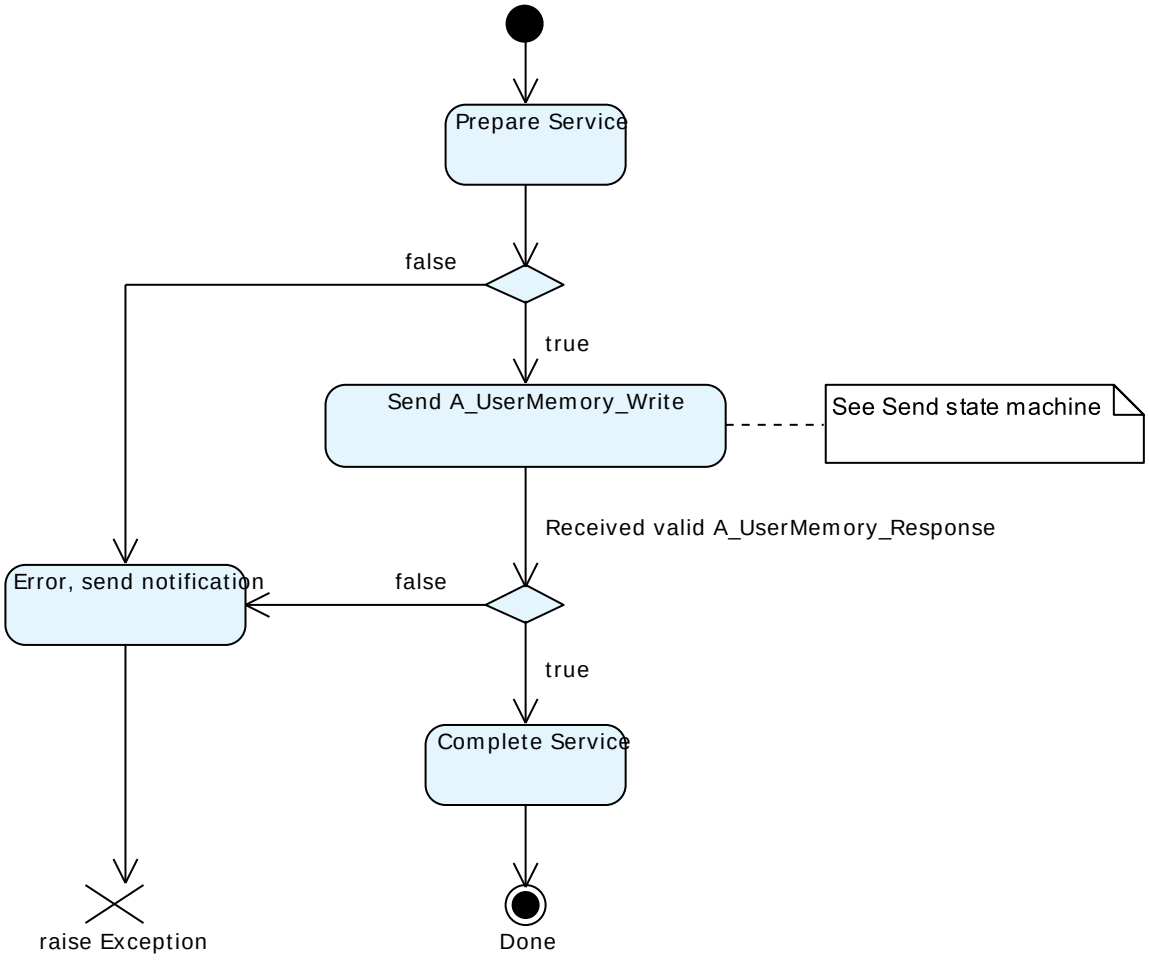
Finite State Machine: UserMemoryRead

3.4.13. UserMemoryWrite

Table 3.16. UserMemoryWrite

Attribute	Description
Name	UserMemoryWrite
Description	Writes data via DMA. If Input Parameter "Verify Response Data" is true then verify the field data in the response with Input Parameter Data. Note: The field number is set to the size from input parameter Data.

Attribute	Description
Inputs	Individual Address (integer) [1 .. 0xFFFF] Address 20 Bit (integer) [0 .. 0xFFFFF] Data (byte array) [1 .. 11 bytes] Verify Response Data (boolean, optional default = true)
Outputs	Address 20 Bit (integer) Number (integer) Data (byte array)
Validation	Via Memory Response
Exceptions	NegativeResponseException. Thrown when a response is received with Number = 0. For example: The requested address space is unreachable or protected or an illegal number of octets are requested. DeviceErrorException. Thrown when either the Address, Number or the Data (only if input parameter "Verify Response Data" is true) in the response do not match those in the request.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.5.6.3 A_UserMemory_Write-service

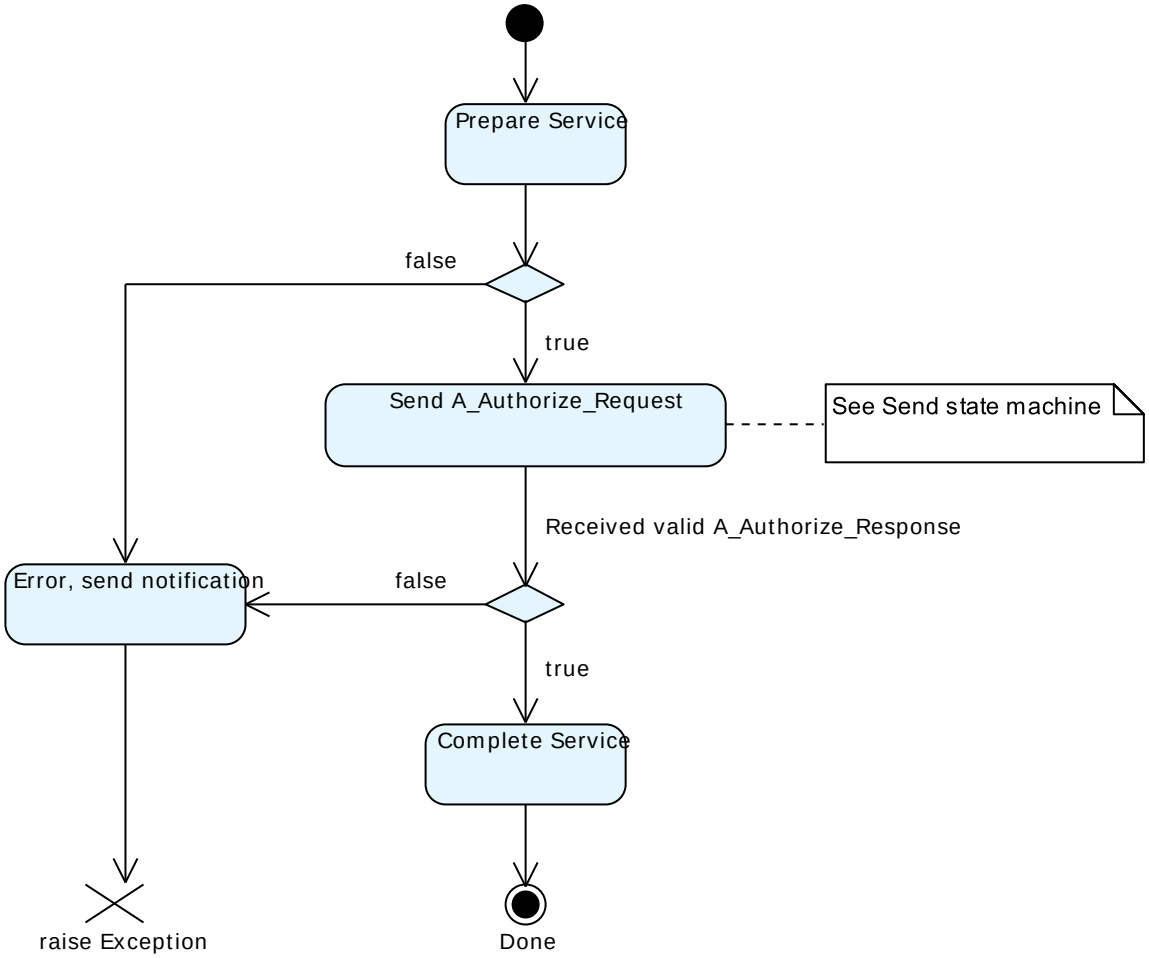


Finite State Machine: UserMemoryWrite

3.4.14. AuthorizeRequest

Table 3.17. AuthorizeRequest

Attribute	Description
Name	AuthorizeRequest
Description	Sends an authorize request
Inputs	Individual Address (integer) [1 .. 0xFFFF] Key (integer) [0 .. 0xFFFFFFFF]
Outputs	Level (integer)
Validation	Via Service Response
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.5.7 A_Authorize_Request-service



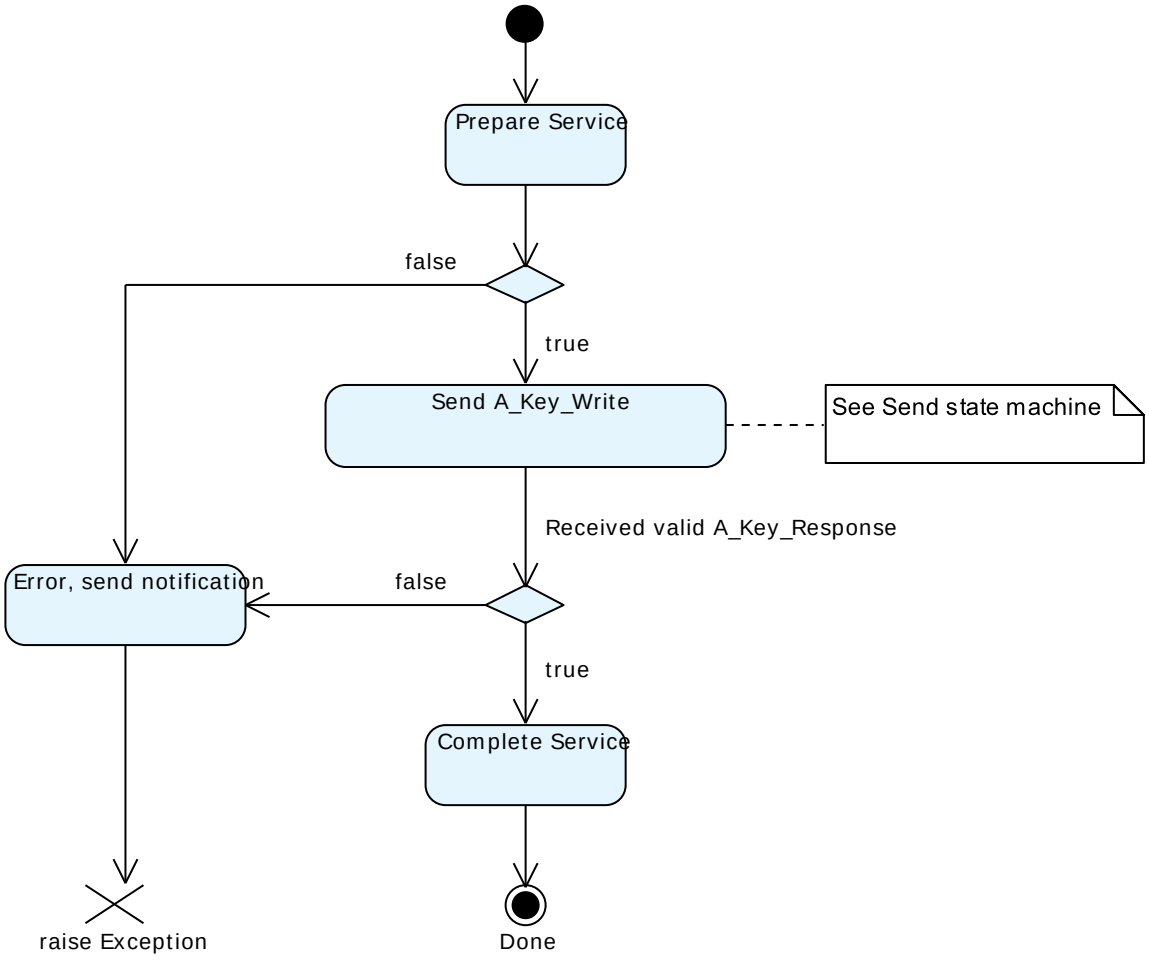
Finite State Machine: AuthorizeRequest

3.4.15. KeyWrite

Table 3.18. KeyWrite

Attribute	Description
Name	KeyWrite
Description	Writes an authorization key

Attribute	Description
Inputs	Individual Address (integer) [1 .. 0xFFFF] Level (integer) [0 .. 0xFF] Key (integer) [0 .. 0xFFFFFFFF]
Outputs	Level (integer)
Validation	Via Service Response
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.5.8 A_Key_Write-service



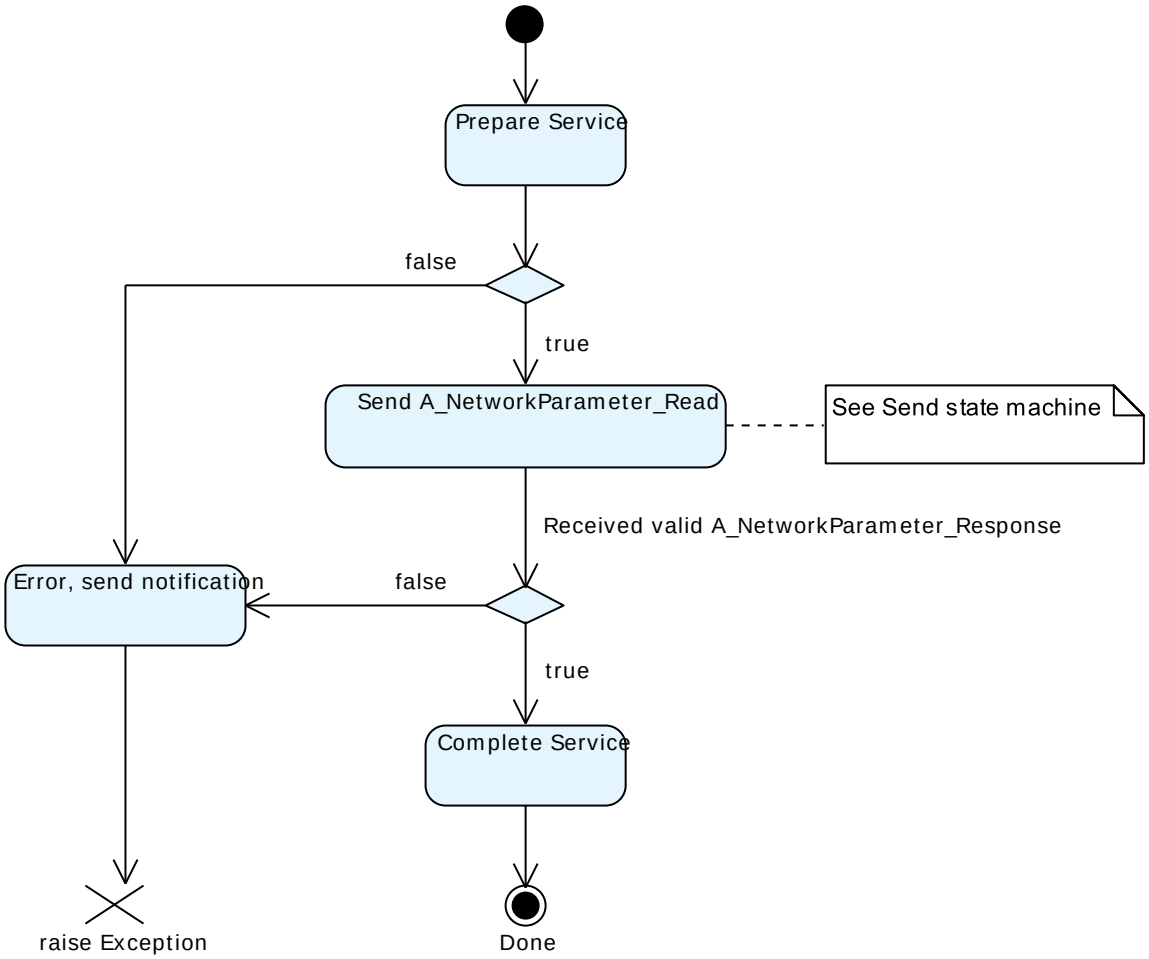
Finite State Machine: KeyWrite

3.4.16. NetworkParameterReadDevice

Table 3.19. NetworkParameterReadDevice

Attribute	Description
Name	NetworkParameterReadDevice
Description	Reads an NetworkParameter from a device.
Inputs	Individual Address (integer) [1 .. 0xFFFF] ObjectType (integer) [0 .. 0xFFFF] Property ID (integer) [1 .. 0xFF] Test Info (byte array) [1 .. 11 bytes]

Attribute	Description
Outputs	ObjectType (integer) Property ID (integer) Test Info and Result (byte array)
Validation	Via Service Response
Exceptions	ObjectTypeNotSupportedException. Thrown when in the response the field test_info and test_result not exists, the object type is 0xFFFF and the property id is 0xFF. i.e: the requested object type does not exist. PropertyIdNotSupportedException. Thrown when in the response the field test_info and test_result not exist, the object type is the requestes object type and the property id is 0xFF. i.e: the requested property does not exist. NegativeResponseException. Thrown when the test_info and test_result fields are not present. DeviceErrorException. Thrown when the Object Type and/or Property Id of the request do not match those in the response.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.2.6 A_NetworkParameter_Read-service AN124 v03 Interface Object Index Discovery AS

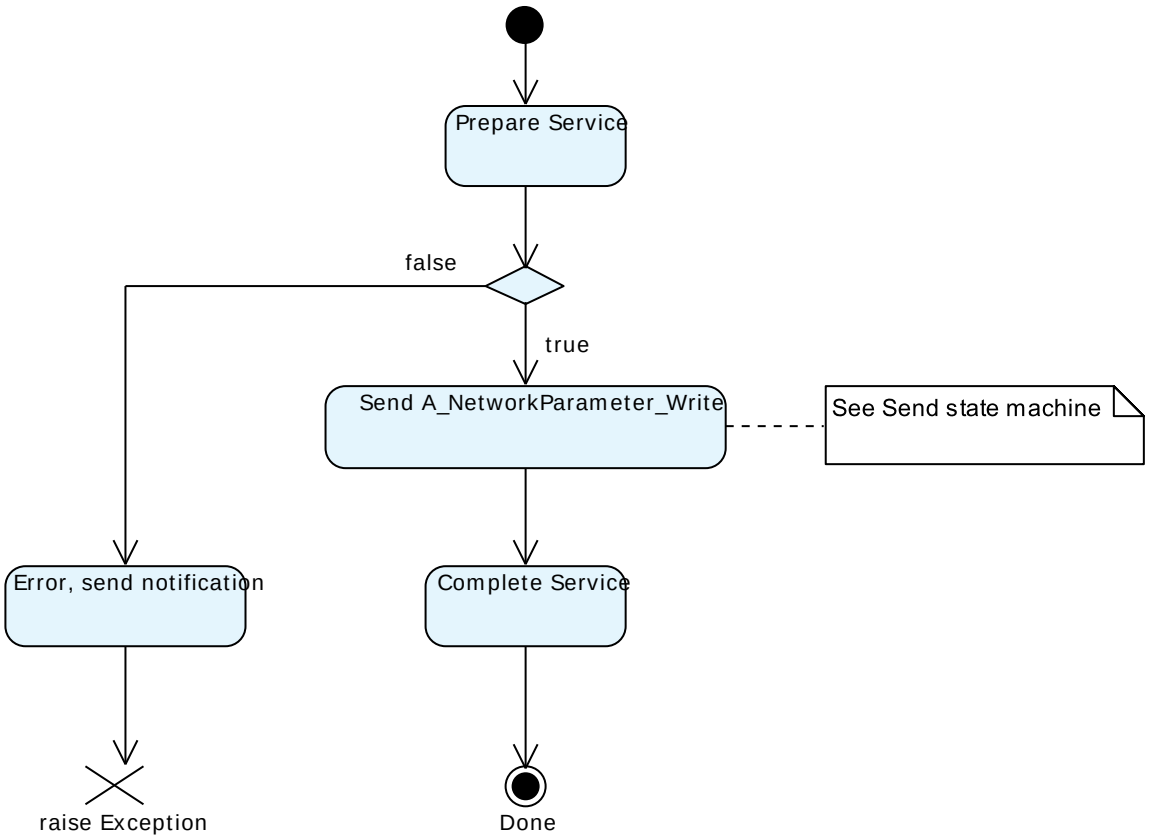


Finite State Machine: NetworkParameterReadDevice

3.4.17. NetworkParameterWriteDevice

Table 3.20. NetworkParameterWriteDevice

Attribute	Description
Name	NetworkParameterWriteDevice
Description	Writes an NetworkParameter to a device.
Inputs	Individual Address (integer) [1 .. 0xFFFF] ObjectType (integer) [0 .. 0xFFFF] Property ID (integer) [1 .. 0xFF] Value (byte array) [1 .. 11 bytes]
Outputs	No Outputs
Validation	None
Connection Status (default)	Connection-less
Specification	AN134 v02 Flexible E-Mode Channels DV



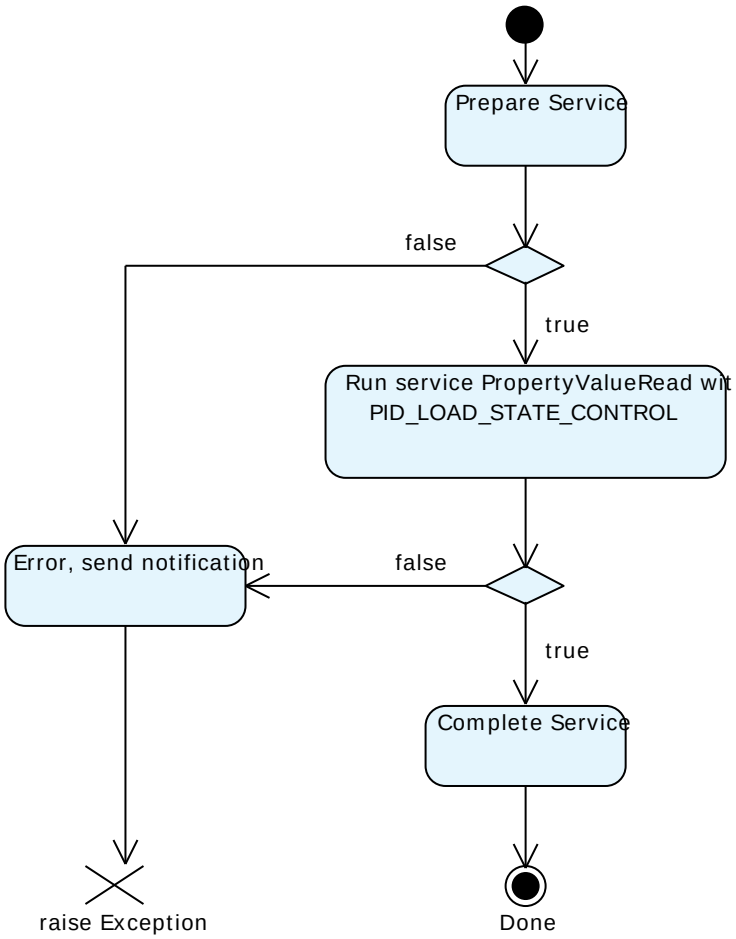
Finite State Machine: NetworkParameterWriteDevice

3.4.18. LoadStateRead

Table 3.21. LoadStateRead

Attribute	Description
Name	LoadStateRead

Attribute	Description
Description	Reads the load state via *Section 3.4.6, "PropertyValueRead"* with Property Id PID_LOAD_STATE_CONTROL
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [1 .. 0xFF]
Outputs	Load State (integer)
Validation	Via Service Response
Exceptions	DeviceErrorException. Thrown when the data length in the read response is not 1. Exceptions from service PropertyValueRead.
Connection Status (default)	Connection-less
Specification	03_05_01 Resources v1.2 AS.pdf 4.16.2 Load State Machine – Realisation Type 1 (Property based)



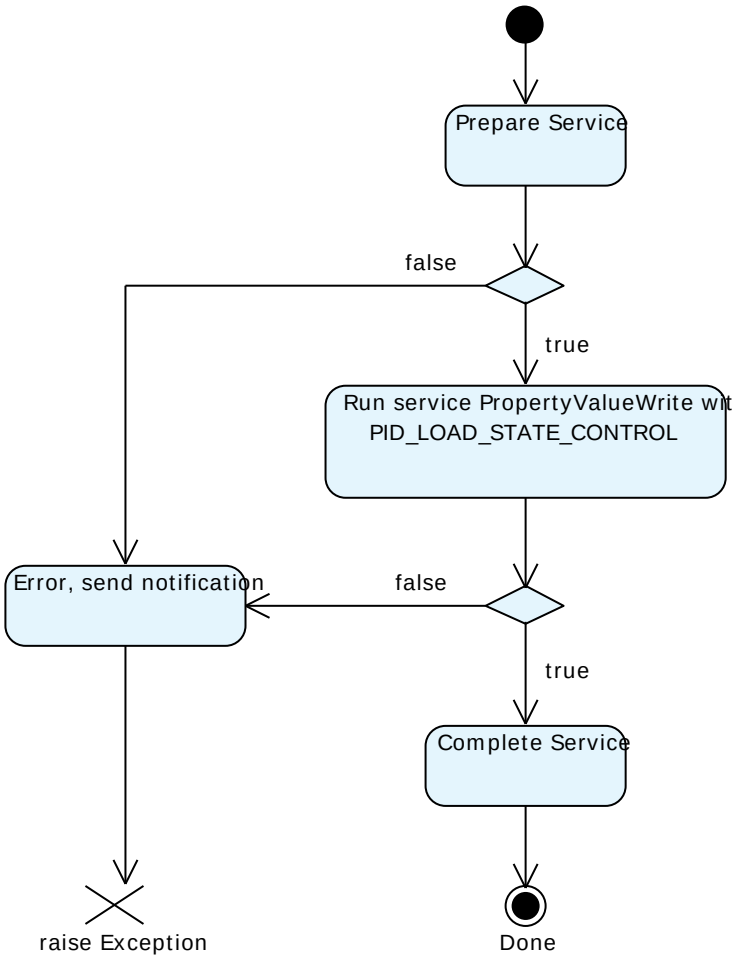
Finite State Machine: LoadStateRead

3.4.19. LoadStateWrite

Table 3.22. LoadStateWrite

Attribute	Description
Name	LoadStateWrite

Attribute	Description
Description	Writes the load state via *Section 3.4.7, "PropertyValueWrite"* with Property Id PID_LOAD_STATE_CONTROL. Note: The bytes for the additional information will fill up with 0.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [1 .. 0xFF] Load Event (integer) [1 .. 0xFF] Additional Information (byte array, optional default = 00 00 00 00 00 00 00 00 00) [0 .. 9 bytes]
Outputs	Load State (integer)
Validation	Via Service Response
Exceptions	DeviceErrorException. Thrown when the data length in the read response is not 1. Exceptions from service PropertyValueWrite.
Connection Status (default)	Connection-less
Specification	03_05_01 Resources v1.2 AS.pdf 4.16.2 Load State Machine – Realisation Type 1 (Property based)

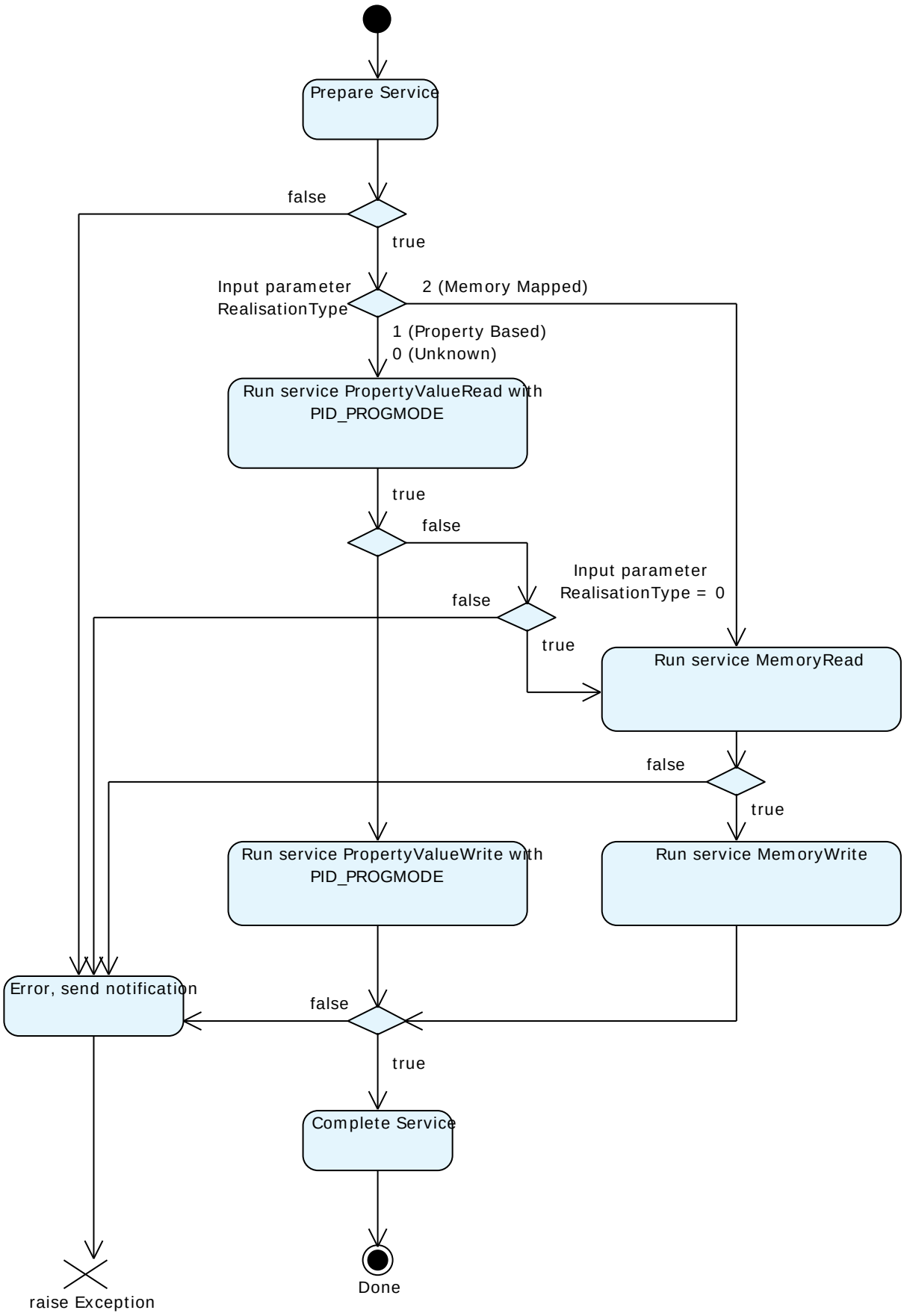


Finite State Machine: LoadStateWrite

3.4.20. SwitchProgMode

Table 3.23. SwitchProgMode

Attribute	Description
Name	SwitchProgMode
Description	Enable / Disable the programming mode (switches the learn LED). Reads the current value writes the new value. The reserved bits will not changed. If Input Parameter RealisationType is ProgModeRealisationType::PropertyBased (=1), than read it via service *Section 3.4.6, "PropertyValueRead"* and write it via service *Section 3.4.7, "PropertyValueWrite"* (PID_PROGMODE). If Input Parameter RealisationType is ProgModeRealisationType::MemoryMapped (=2), than read it via service *Section 3.4.10, "MemoryRead"* and write it via service *Section 3.4.11, "MemoryWrite"* (memory address 0x0060). If Input Parameter RealisationType is ProgModeRealisationType::Unknown (=0), than try it Property-Based and if this was failed than try it MemoryMapped.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Enable ProgMode (boolean) RealisationType (integer, optional default = 0) [0 = Unknown, 1 = PropertyBased, 2 = MemoryMapped]
Outputs	No outputs
Validation	Via Service Response
Exceptions	DeviceErrorException. Thrown when the data length in the read response is not 1. Exceptions from services PropertyValueRead, PropertyValueWrite, MemoryRead and MemoryWrite.
Connection Status (default)	Connection-less
Specification	03_05_01 Resources v1.2 AS 4.3.5 PID_PROGMODE (PID = 54) page 37 4.19 Programming Mode (prog_mode) 06 Profiles v1.8 4.4.1 Programming Mode

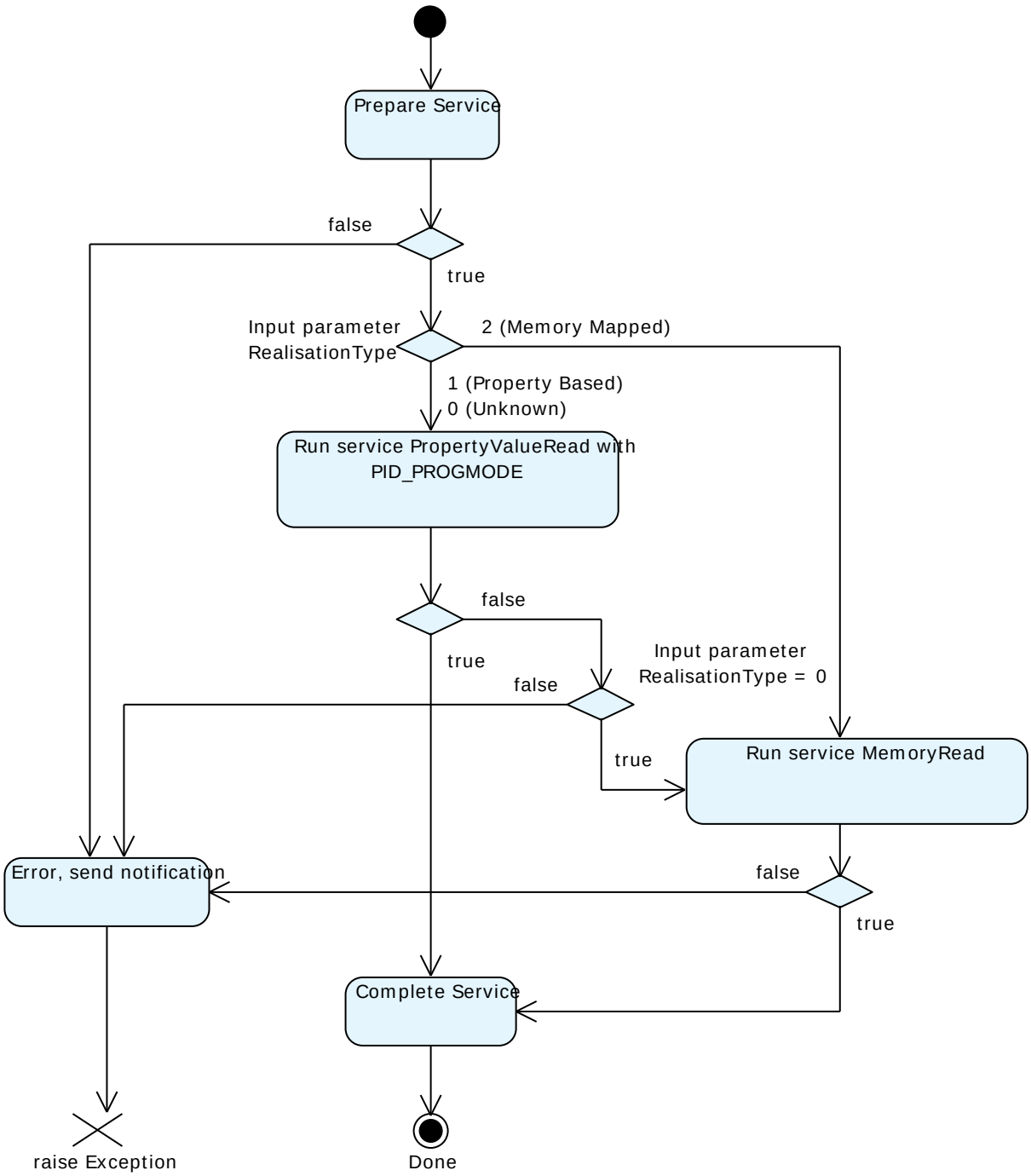


Finite State Machine: SwitchProgMode

3.4.21. ReadProgMode

Table 3.24. ReadProgMode

Attribute	Description
Name	ReadProgMode
Description	Reads the current programming mode. If Input Parameter RealisationType is ProgModeRealisationType::PropertyBased (=1), then read it via service *Section 3.4.6, "PropertyValueRead"* (PID_PROGMODE). If Input Parameter RealisationType is ProgModeRealisationType::MemoryMapped (=2), then read it via service *Section 3.4.10, "MemoryRead"* (memory address 0x0060). If Input Parameter RealisationType is ProgModeRealisationType::Unknown (=0), then try it PropertyBased and if this was failed then try it MemoryMapped.
Inputs	Individual Address (integer) [1 .. 0xFFFF] RealisationType (integer, optional default = 0) [0 = Unknown, 1 = PropertyBased, 2 = MemoryMapped]
Outputs	Is the ProgMode enabled (boolean)
Validation	Via Service Response
Exceptions	DeviceErrorException. Thrown when the data length in the read response is not 1. Exceptions from service PropertyValueRead or MemoryRead.
Connection Status (default)	Connection-less
Specification	03_05_01 Resources v1.2 AS 4.3.5 PID_PROGMODE (PID = 54) page 37 4.19 Programming Mode (prog_mode) 06 Profiles v1.8 4.4.1 Programming Mode



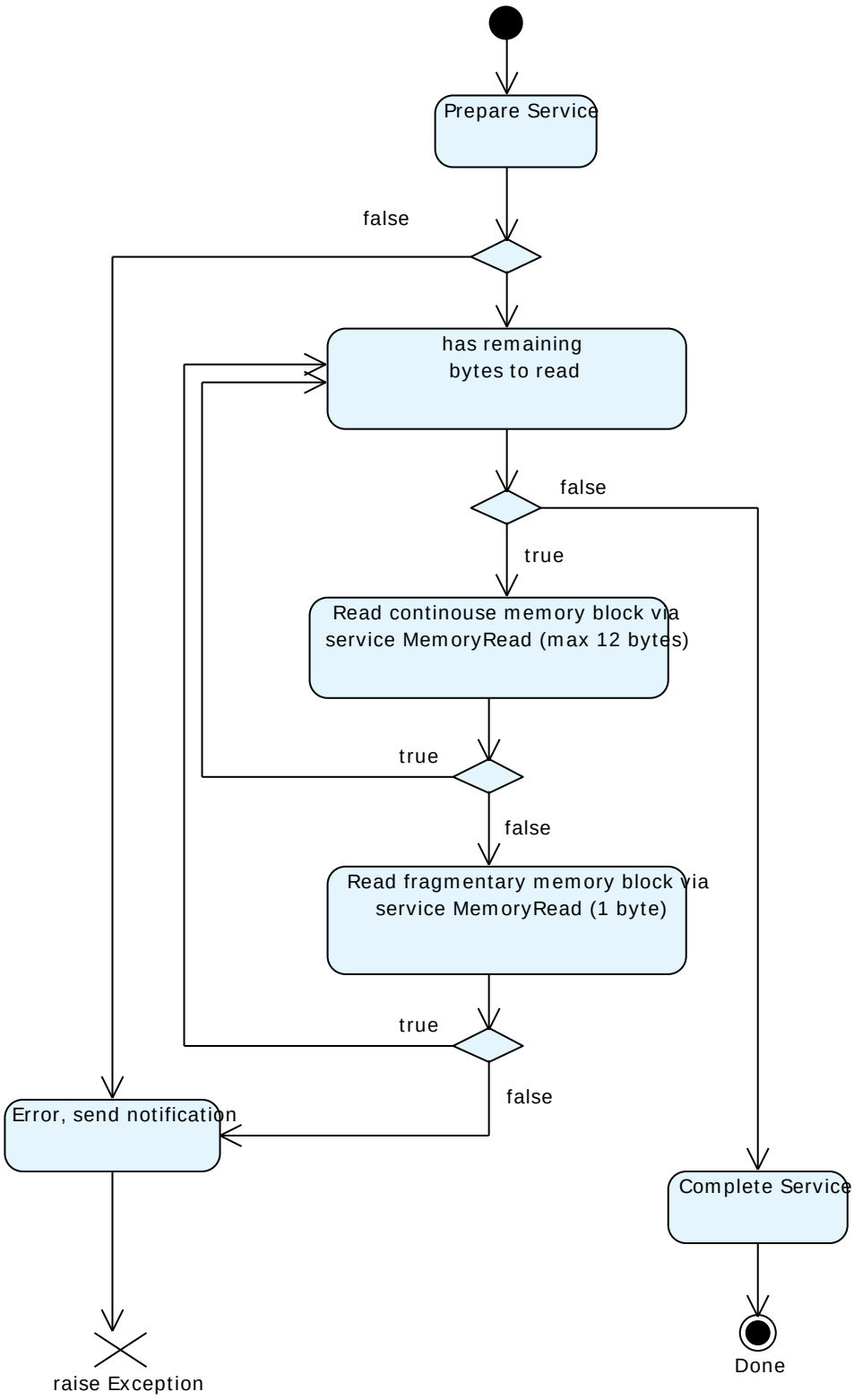
Finite State Machine: ReadProgMode

3.4.22. MemoryBlockRead

Table 3.25. MemoryBlockRead

Attribute	Description
Name	MemoryBlockRead
Description	Reads a contiguous memory block via DMA. The service supports the reading of blocks where the exact start and end address in unknown. For example, we want to read from 0x4000 and read 512 bytes. The following cases are possible:

Attribute	Description
	• The entire memory range is valid. Output parameter Address of 0x4000, data buffer (size is 512). • The start address and less than 512 bytes are valid (for example, 32 bytes). Output parameter Address of 0x4000 and with the corresponding buffer (size is 32 bytes). Note: There could be additional valid memory blocks in the range 0x4020 to 0x4200 and these can be "discovered" with subsequent reads, see next case. • The start address is not valid. Return the first valid memory block in the range. • There are no valid memory blocks in the range. Return empty buffer. The return address in this case is undefined.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Address 16 Bit (integer) [0 .. 0xFFFF] Number (integer) [1 ..]
Outputs	Address 16 Bit (integer) Data (byte array)
Validation	Via Memory Response
Exceptions	Exceptions from service MemoryRead.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.5.3 A_MemoryRead-Service



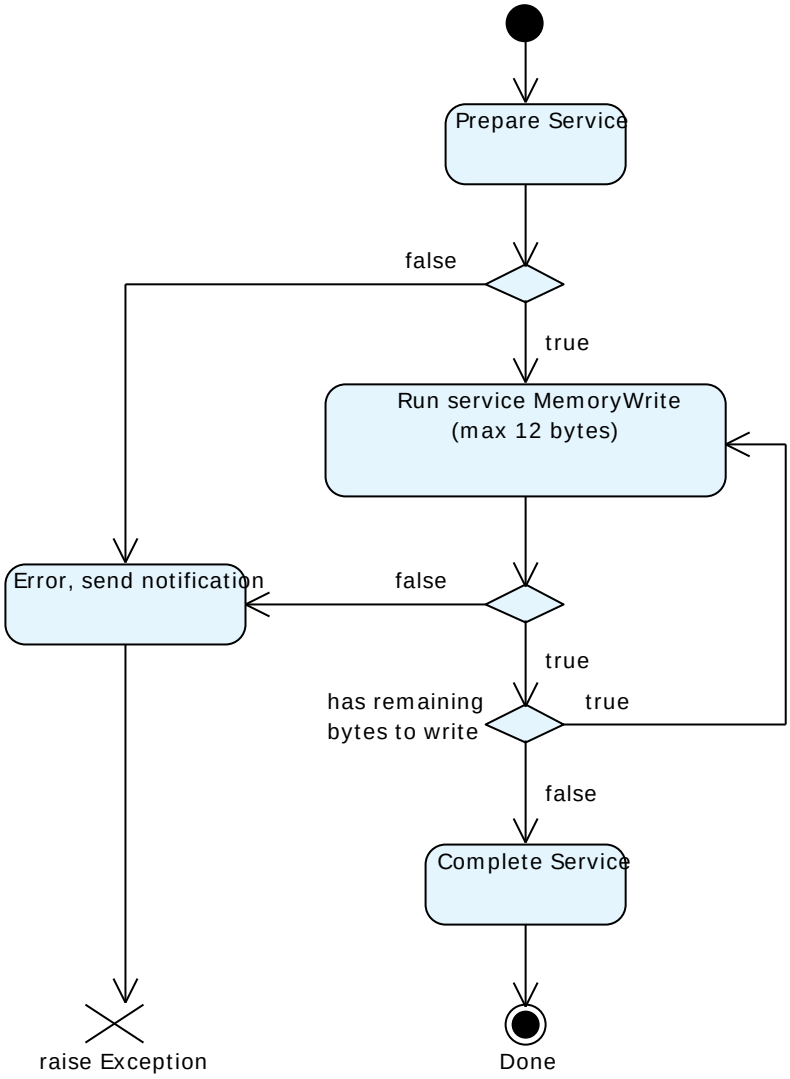
Finite State Machine: MemoryBlockRead

3.4.23. MemoryBlockWrite

Table 3.26. MemoryBlockWrite

Attribute	Description
Name	MemoryBlockWrite

Attribute	Description
Description	Writes a contiguous memory block via DMA.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Address 16 Bit (integer) [0 .. 0xFFFF] Data (byte array) [1 .. bytes] DeviceSupportsVerifyMode (bool, optional default = false) Verify Response Data (boolean, optional default = true)
Outputs	Address 16 Bit (integer) Number (integer) Data (byte array)
Validation	Via Memory Response
Exceptions	Exceptions from service MemoryWrites.
Connection Status (default)	Connection-less
Specification	03_03_07 Application Layer 3.5.4 A_Memory_Write-service

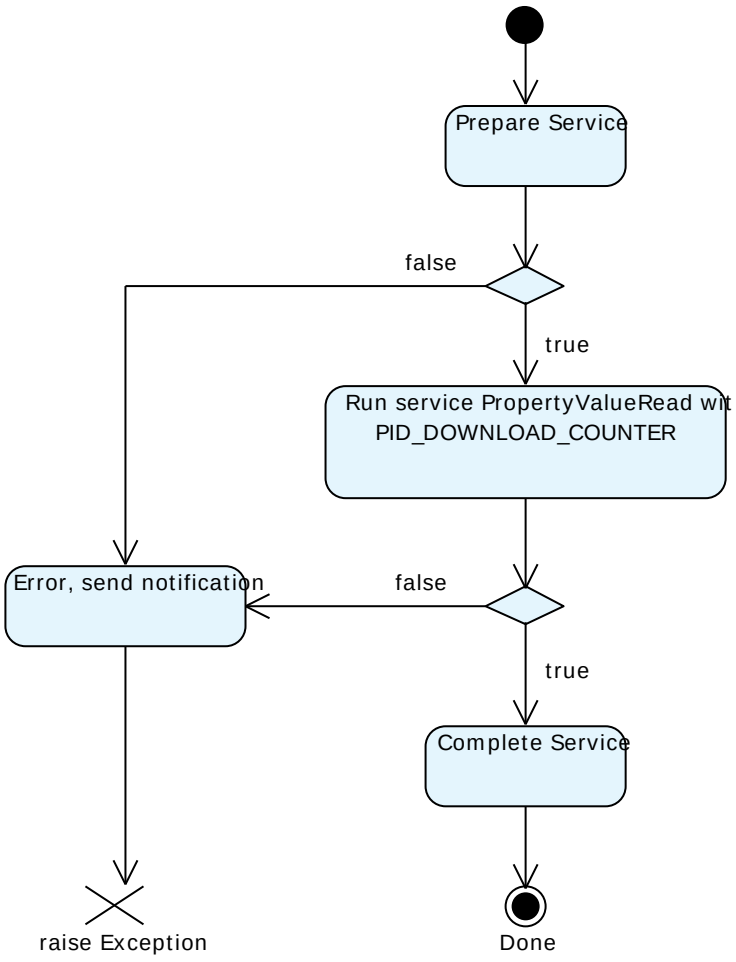


Finite State Machine: MemoryBlockWrite

3.4.24. ReadDownloadCounter

Table 3.27. ReadDownloadCounter

Attribute	Description
Name	ReadDownloadCounter
Description	Reads the download counter via service *Section 3.4.6, "PropertyValueRead"*
Inputs	Individual Address (integer) [1 .. 0xFFFF]
Outputs	Download Counter (integer)
Validation	Via Service Response
Exceptions	DeviceErrorException. Thrown when the data length in the read response is not 2. Exceptions from service PropertyValueRead
Connection Status (default)	Connection-less
Specification	AN137 v01 Configuration Signature DP

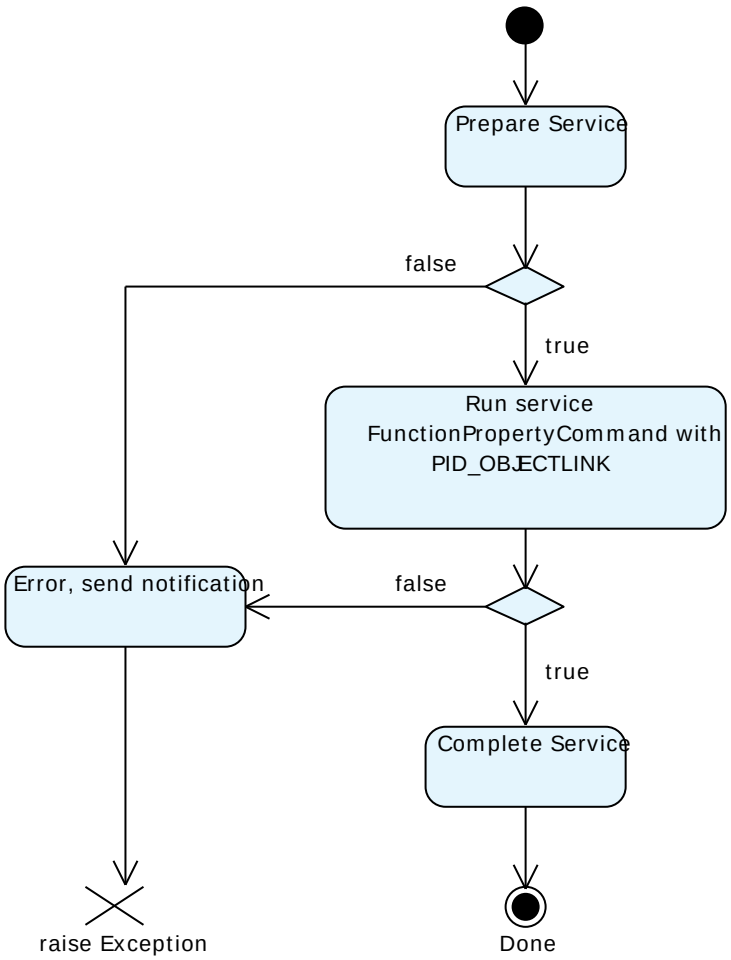


Finite State Machine: ReadDownloadCounter

3.4.25. WriteObjectLink

Table 3.28. WriteObjectLink

Attribute	Description
Name	WriteObjectLink
Description	This service establishs or breaks a single link between a Group Object and a Group Address or an Extended Group Address. It uses the service *Section 3.4.8, "FunctionPropertyCommand"*. Use the service *Section 3.4.27, "ObjectIndexRead"* with Object Type = E-Mode Channel Object, to get the Object Index of the channel. The handle is the group object index within the E-Mode Channel. The GO-numbering within the E-Mode Channel shall start with 1. Note: This service is only for FEC devices. It has a other frame format as for SEC devices, which use the same PID_OBJECTLINK in in the Device Object.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [1 .. 0xFF] Should delete link (boolean) false = add link, true = delete link Is sending (boolean) (set to false, if Should delete link is true) Serial Number (6 byte KNX serial number, optional default = 00 00 00 00 00 00) [6 bytes] Group Address (integer) [1 .. 0xFFFF] Handle (integer) [1 .. 0xFFFF]
Outputs	Return Code (integer)
Validation	Via Service Response
Exceptions	Exceptions from service FunctionPropertyCommand.
Connection Status (default)	Connection-less
Specification	AN134 v02 Flexible E-Mode Channels DV 2.3.4.13.1 Function Write Object Link



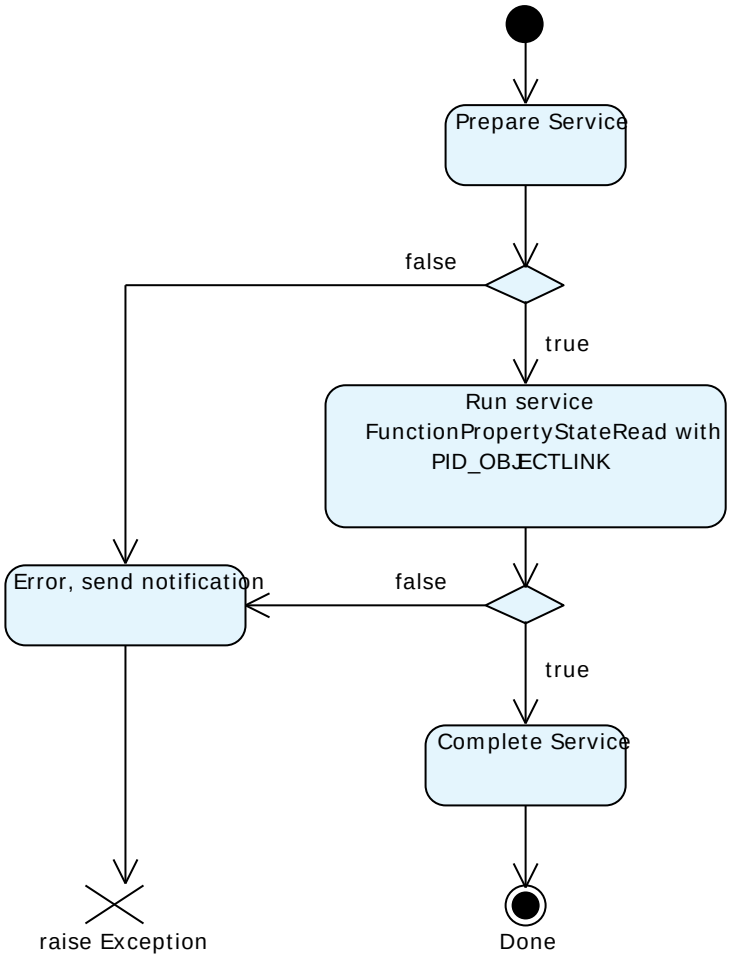
Finite State Machine: WriteObjectLink

3.4.26. ReadObjectLink

Table 3.29. ReadObjectLink

Attribute	Description
Name	ReadObjectLink
Description	This service read a single Group Addresses or Extended Group Address linked to a single Group Object. It uses the service *Section 3.4.9, "FunctionPropertyStateRead"*. Use the service *Section 3.4.27, "ObjectIndexRead"* with Object Type = E-Mode Channel Object, to get the Object Index of the channel. The handle is the group object index within the E-Mode Channel. The GO-numbering within the E-Mode Channel shall start with 1. Note: This service is only for FEC devices. It has a other frame format as for SEC devices, which use the same PID_OBJECTLINK in in the Device Object.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Object Index (integer) [1 .. 0xFF] Handle (integer) [1 .. 0xFFFF] Iterator (integer) [0 .. 0xFF]
Outputs	Object Index (integer) Is sending (boolean) Serial Number (6 byte KNX serial number)

Attribute	Description
	Group Address (integer) Handle (integer) Return Code (integer)
Validation	Via Service Response
Exceptions	Exceptions from service FunctionPropertyStateRead and DeviceErrorException if the telegram is to short.
Connection Status (default)	Connection-less
Specification	AN134 v02 Flexible E-Mode Channels DV 2.3.4.13.2 Function Read Object Link



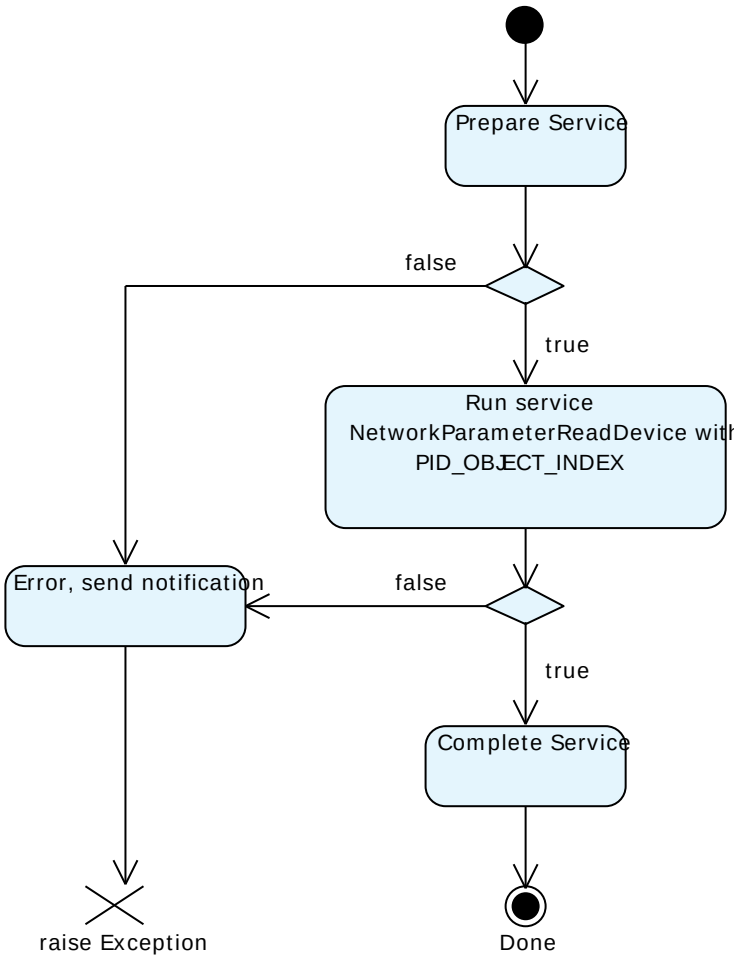
Finite State Machine: ReadObjectLink

3.4.27. ObjectIndexRead

Table 3.30. ObjectIndexRead

Attribute	Description
Name	ObjectIndexRead
Description	This services is used to find the Object Index(es) of one Object Type in a device. This service uses the service *Section 3.4.16, "NetworkParameterReadDevice"*
Inputs	Individual Address (integer) [1 .. 0xFFFF]

Attribute	Description
	Object Type (integer) [0 .. 0xFFFF] Start instance (integer) [0 .. 0xFF] Number of instances (integer) [0 .. 0xFF]
Outputs	Object Type (integer) Start instance (integer) Number of instances (integer) Object index(es) (byte array)
Validation	Via Service Response
Exceptions	Exceptions from service NetworkParameterReadDevice. DeviceErrorException. Thrown when the ObjectType do not match the associated request parameters.
Connection Status (default)	Connection-less
Specification	AN124 v03 Interface Object Index Discovery AS



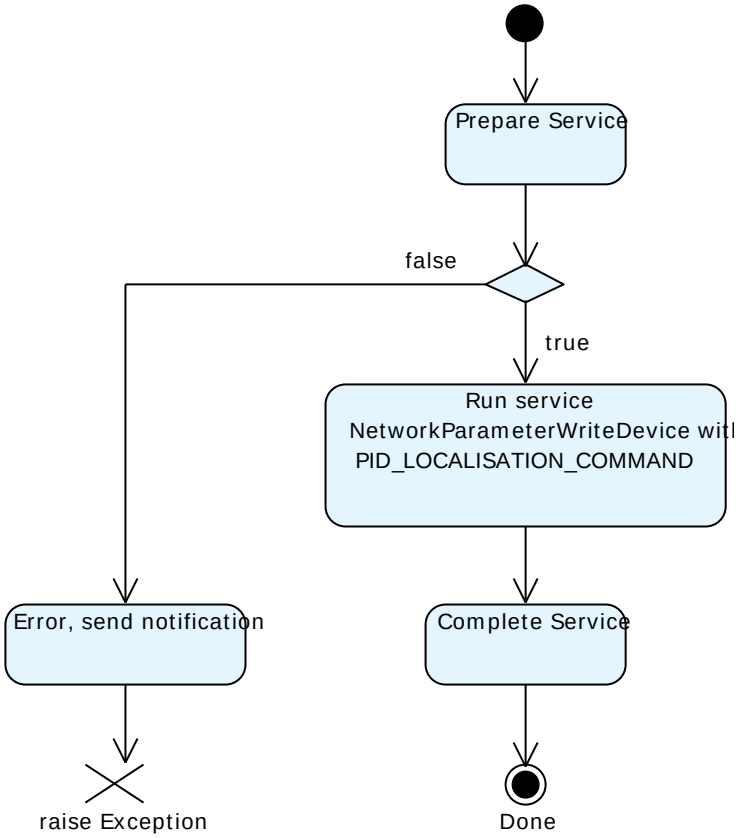
Finite State Machine: ObjectIndexRead

3.4.28. SendLocalisationCommandDevice

Table 3.31. SendLocalisationCommandDevice

Attribute	Description
Name	SendLocalisationCommandDevice

Attribute	Description
Description	Sends a localization command. The easy controller sends it if the user want localize a channel. E.g a LED go on. This service uses the service *Section 3.4.17, "NetworkParameterWriteDevice"*.
Inputs	Individual Address (integer) [1 .. 0xFFFF] Disable Physical Action (boolean) Start (boolean) (0 = stop, 1 = start) Channel Number (integer) [0 .. 0xFFFF] (0 = entire device)
Outputs	No Outputs
Validation	None
Connection Status (default)	Connection-less
Specification	AN134 v02 Flexible E-Mode Channels DV 2.3.7.6 E-Mode Device Object – PID_LOCALISATION_COMMAND (PID = 62)



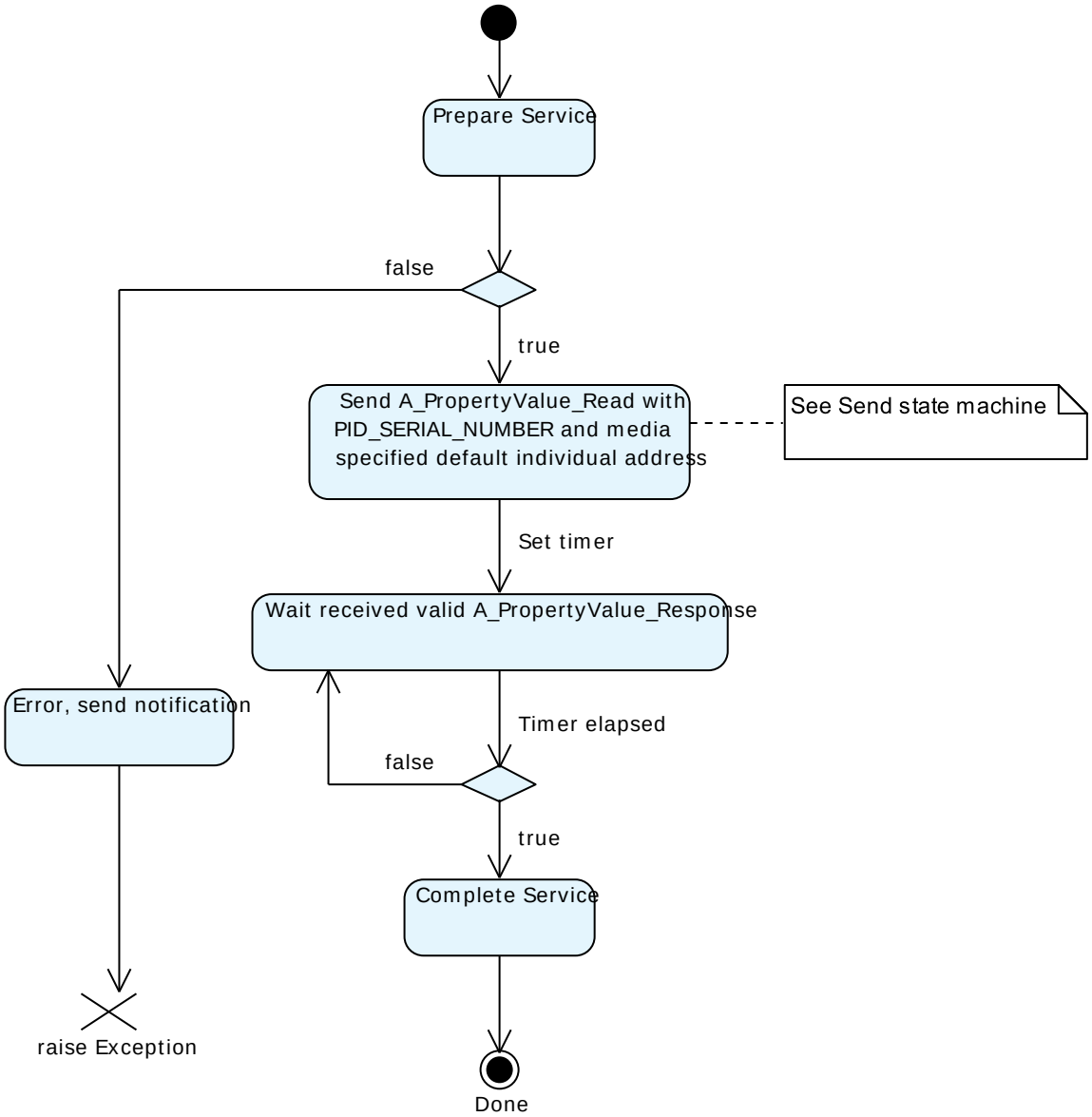
Finite State Machine: SendLocalisationCommandDevice

3.4.29. FindUnconfiguredDevices

Table 3.32. FindUnconfiguredDevices

Attribute	Description
Name	FindUnconfiguredDevices
Description	Searches for devices that are not configured. An unconfigured device has domain address 00 00 00 00 00 00 (RF only) and the individual address is the media specified default individual address.

Attribute	Description
	The input parameter MediumType is coded as DPT_Media. Currently RF and TP1 are supported.
Inputs	MediumType (integer, DPT_Media) [TP1 = 2, RF = 16]
Outputs	SerialNumbers (list of byte array) Wait time in ms (integer, optional default = 500ms)
Validation	None
Exceptions	Exceptions from service PropertyValueRead.
Connection Status (default)	Connection-less
Specification	DPT_Media: 03_07_02 Datapoint Types v1.4 AS 7.3.1 Datapoint Type 'Media'



Finite State Machine: FindUnconfiguredDevices

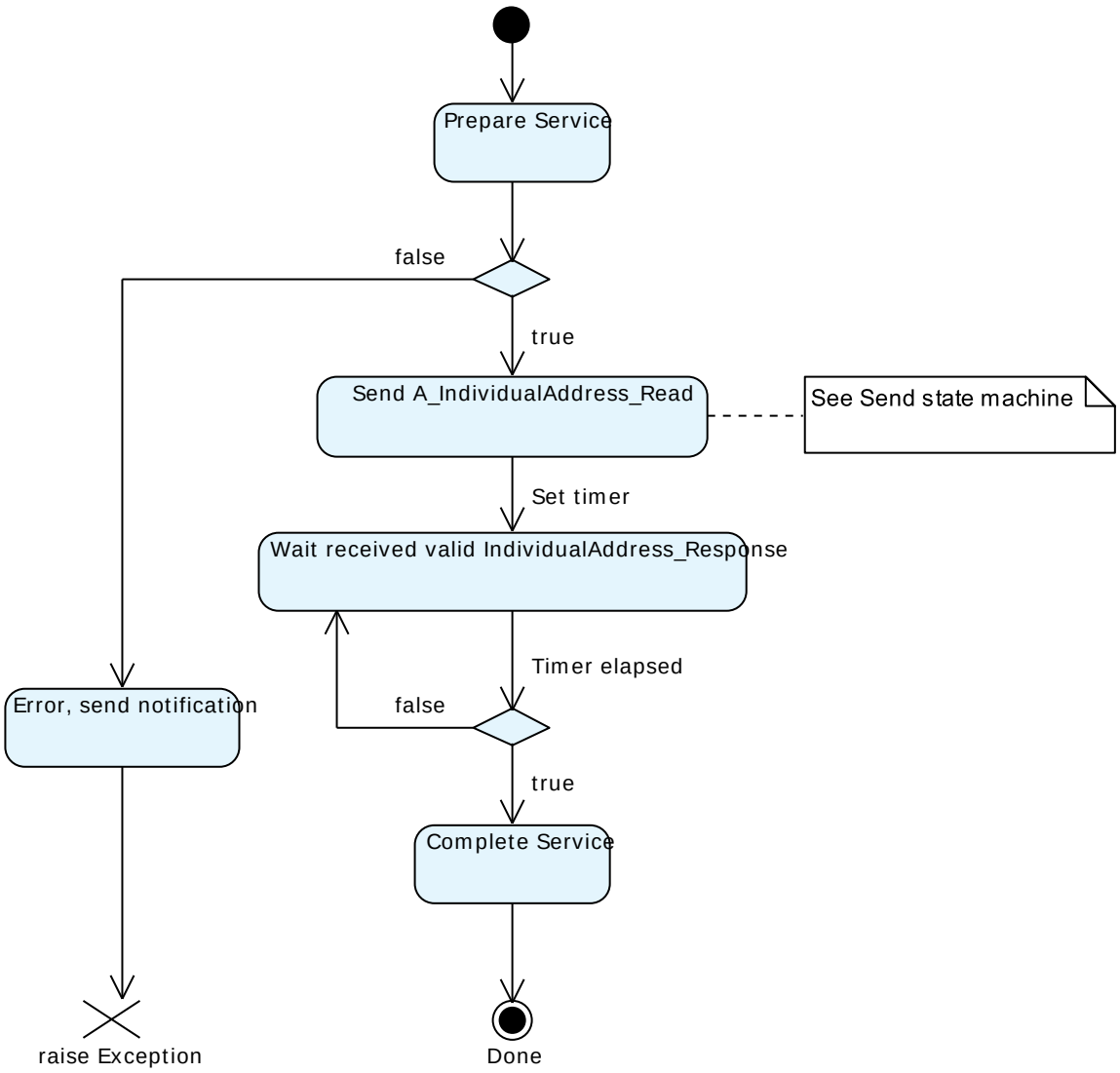
3.5. Broadcast Services

A broadcast service is a service that communicates mainly via KNX broadcast services within the Domain Address of the device. Devices are uniquely identified by either an a Serial Number, or by the programming mode LED. A system broadcast service is a service that communicates mainly via KNX system broadcast services not limited to a Domain. Devices are uniquely identified by either a Serial Number, or by the programming mode LED. The following services are configured to run either as broadcast or system broadcast as required however this can be modified by setting the System Broadcast property of the service.

3.5.1. IndividualAddressProgModeRead

Table 3.33. IndividualAddressProgModeRead

Attribute	Description
Name	IndividualAddressProgModeRead
Description	Reads the Individual Addresses of devices in Programming Mode (i.e. with the Learn LED enabled)
Broadcast Type	Broadcast
Inputs	Wait time in ms (integer, optional default = 500ms)
Outputs	Individual Addresses (list of integer)
Validation	None
Specification	03_03_07 Application Layer 3.2.3 A_IndividualAddress_Read-service



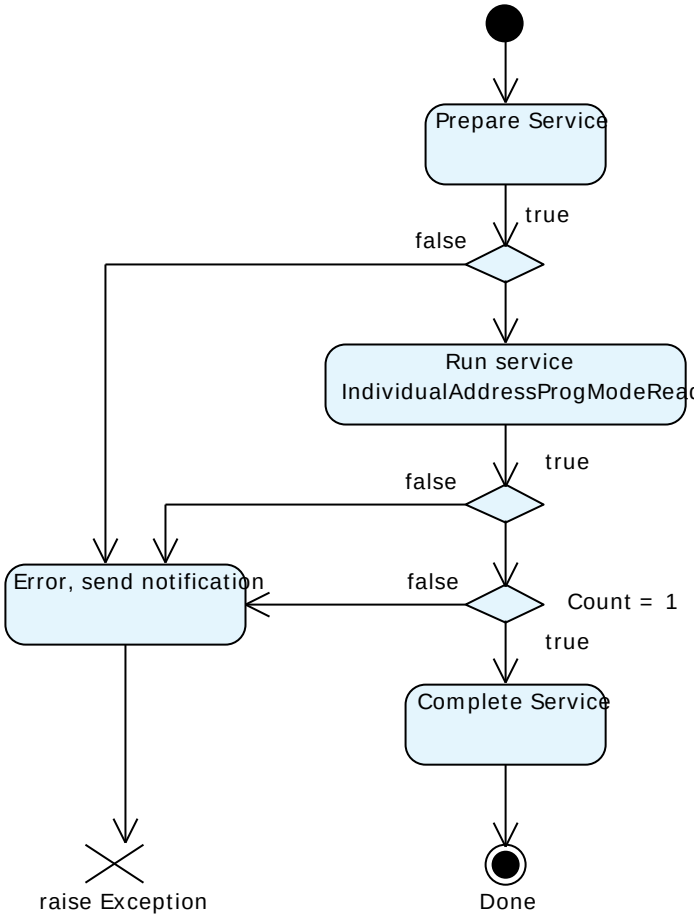
Finite State Machine: IndividualAddressProgModeRead

3.5.2. IndividualAddressProgModeReadSingle

Table 3.34. IndividualAddressProgModeReadSingle

Attribute	Description
Name	IndividualAddressProgModeReadSingle
Description	Reads a single Individual Address of a device in Programming Mode. This service calls the service *Section 3.5.1, "IndividualAddressProgModeRead"* to obtain the list of Individual Addresses. If the list size is equal to 1 the service is successful otherwise an exception is thrown.
Broadcast Type	Broadcast
Inputs	Wait time in ms (integer, optional default = 500ms)
Outputs	IndividualAddress (integer)
Validation	Validates that a single Individual Address is received.
Exceptions	NoDeviceInProgModeException MoreDevicesInProgModeException

Attribute	Description
Specification	03_03_07 Application Layer 3.2.3 A_IndividualAddress_Read-service



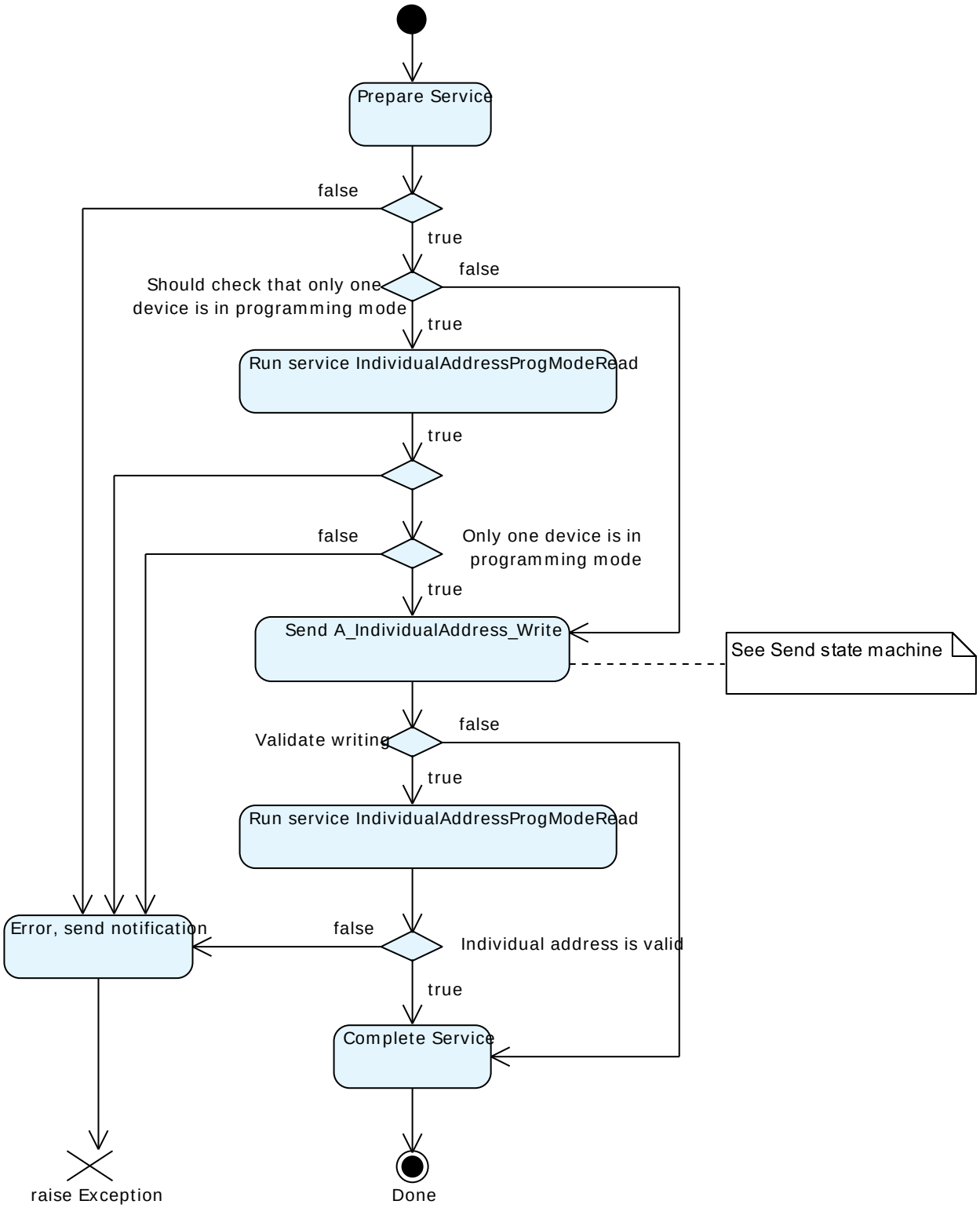
Finite State Machine: IndividualAddressProgModeReadSingle

3.5.3. IndividualAddressProgModeWrite

Table 3.35. IndividualAddressProgModeWrite

Attribute	Description
Name	IndividualAddressProgModeWrite
Description	Writes the specified Individual Address to a device in Programming Mode (i.e. with the Learn LED enabled)
Broadcast Type	Broadcast
Inputs	New Individual Address (integer) [1 .. 0xFFFF] Check only one device is in ProgMode (Boolean, optional default = true) Validate Writing (Boolean, optional default = true) Wait time for reading in ms (integer, optional default = 500ms)
Outputs	No Outputs
Validation	There is only one device in programming mode (optional) Write verify: the individual address was successfully written (optional)

Attribute	Description
Exceptions	NoDeviceInProgModeException MoreDevicesInProgModeException WriteIndAddrFailedException
Specification	03_03_07 Application Layer 3.2.2 A_IndividualAddress_Write-service

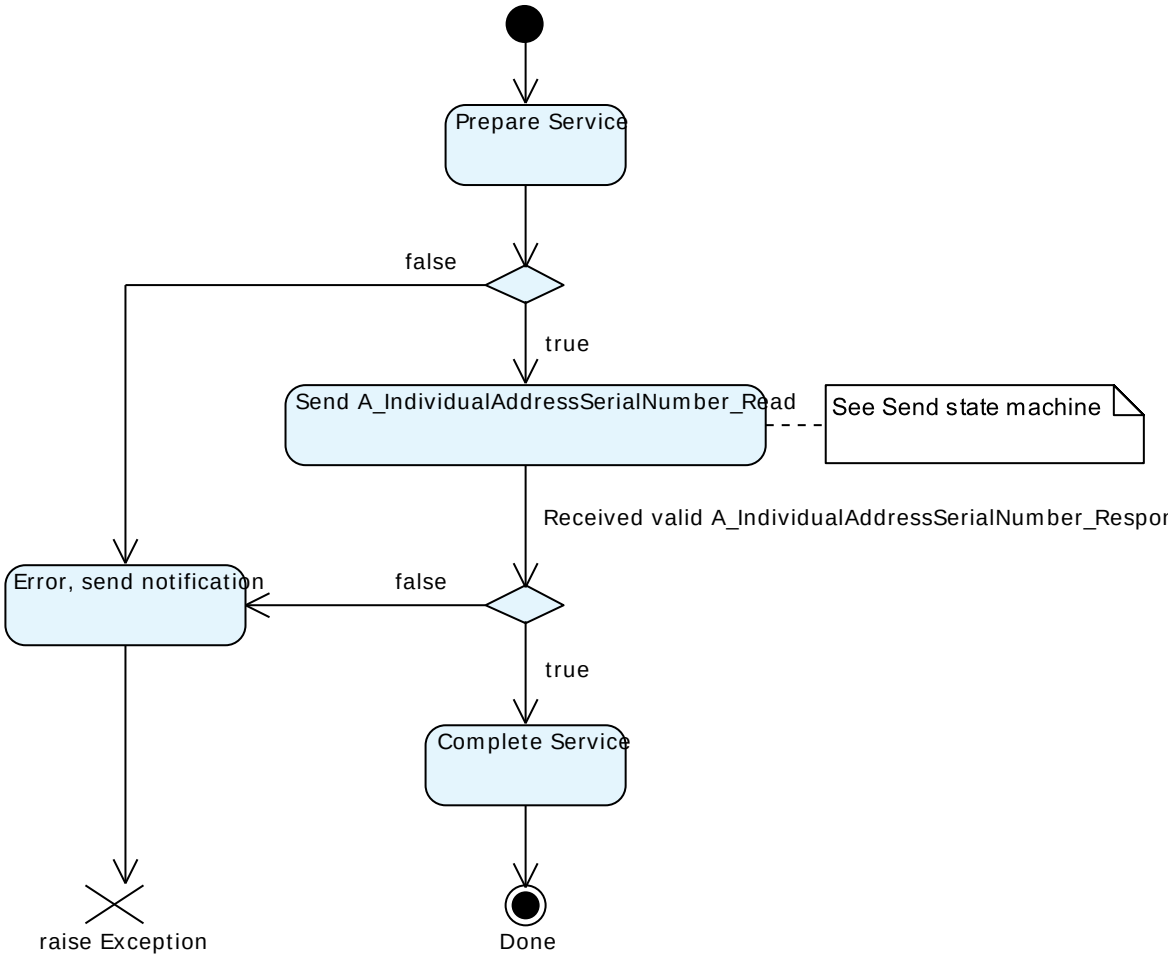


Finite State Machine: IndividualAddressProgModeWrite

3.5.4. IndividualAddressSerialNumberRead

Table 3.36. IndividualAddressSerialNumberRead

Attribute	Description
Name	IndividualAddressSerialNumberRead
Broadcast Type	Broadcast
Description	Reads the Individual Address of a device that matches the specified Serial Number
Inputs	Serial Number (6 byte KNX serial number)
Outputs	Individual Address (integer) [1 .. 0xFFFF] Serial Number (6 byte KNX serial number) [6 bytes]
Validation	None
Exceptions	DeviceErrorException. Thrown when the Serial Number do not match the associated request parameters.
Specification	03_03_07 Application Layer 3.2.4 A_IndividualAddressSerialNumber_Read-service

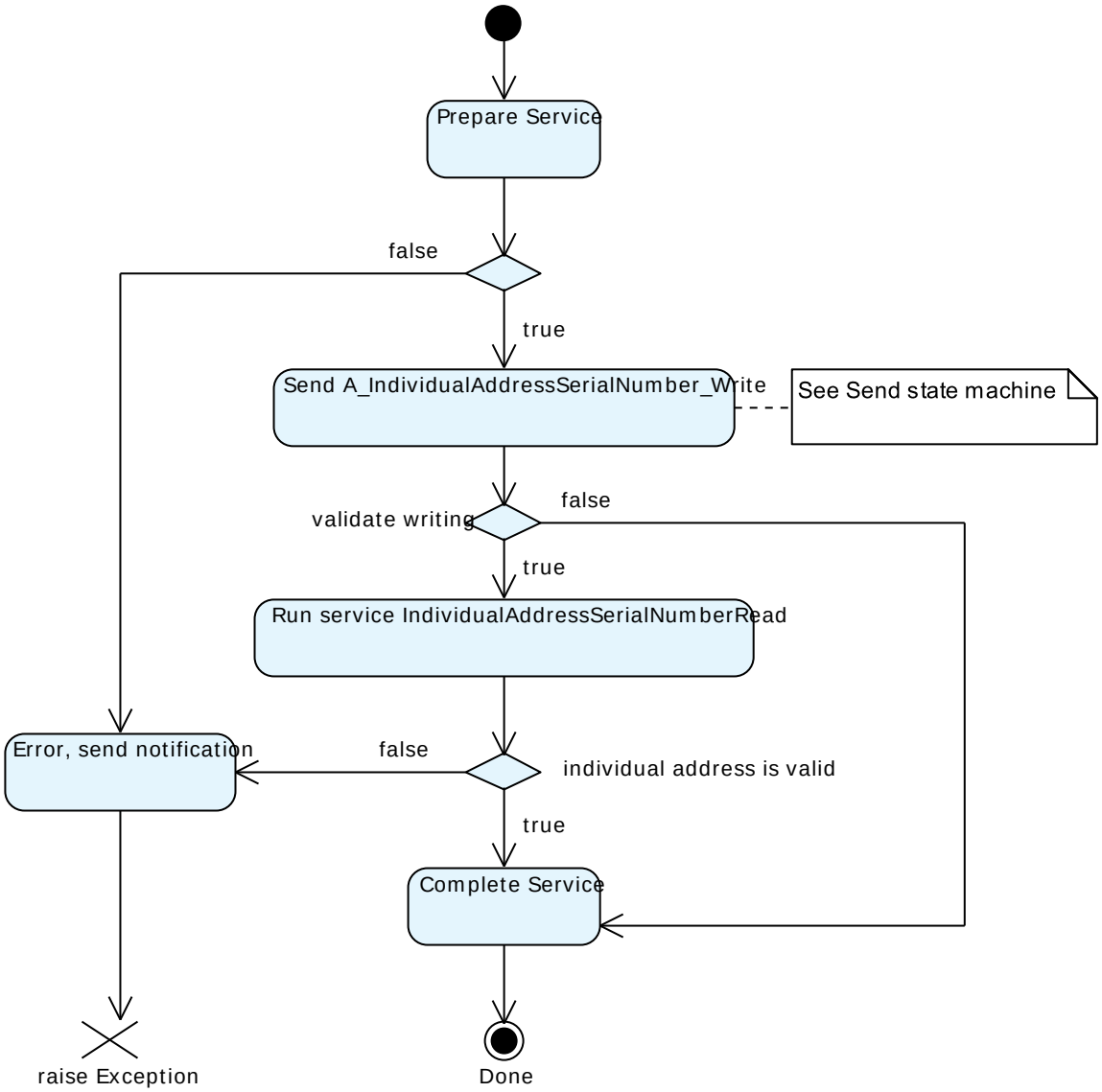


Finite State Machine: IndividualAddressSerialNumberRead

3.5.5. IndividualAddressSerialNumberWrite

Table 3.37. IndividualAddressSerialNumberWrite

Attribute	Description
Name	IndividualAddressSerialNumberWrite
Broadcast Type	Broadcast
Description	Writes the Individual Address of a device
Inputs	Individual Address (integer) [1 .. 0xFFFF] Serial Number (byte array) [6 bytes] Validate (boolean, optional default = true)
Outputs	No Outputs
Validation	Write verify: the individual address was successfully written (optional)
Exceptions	WriteIndAddrFailedException Exceptions from service IndividualAddressSerialNumberRead.
Specification	03_03_07 Application Layer 3.2.5 A_IndividualAddressSerialNumber_Write-service

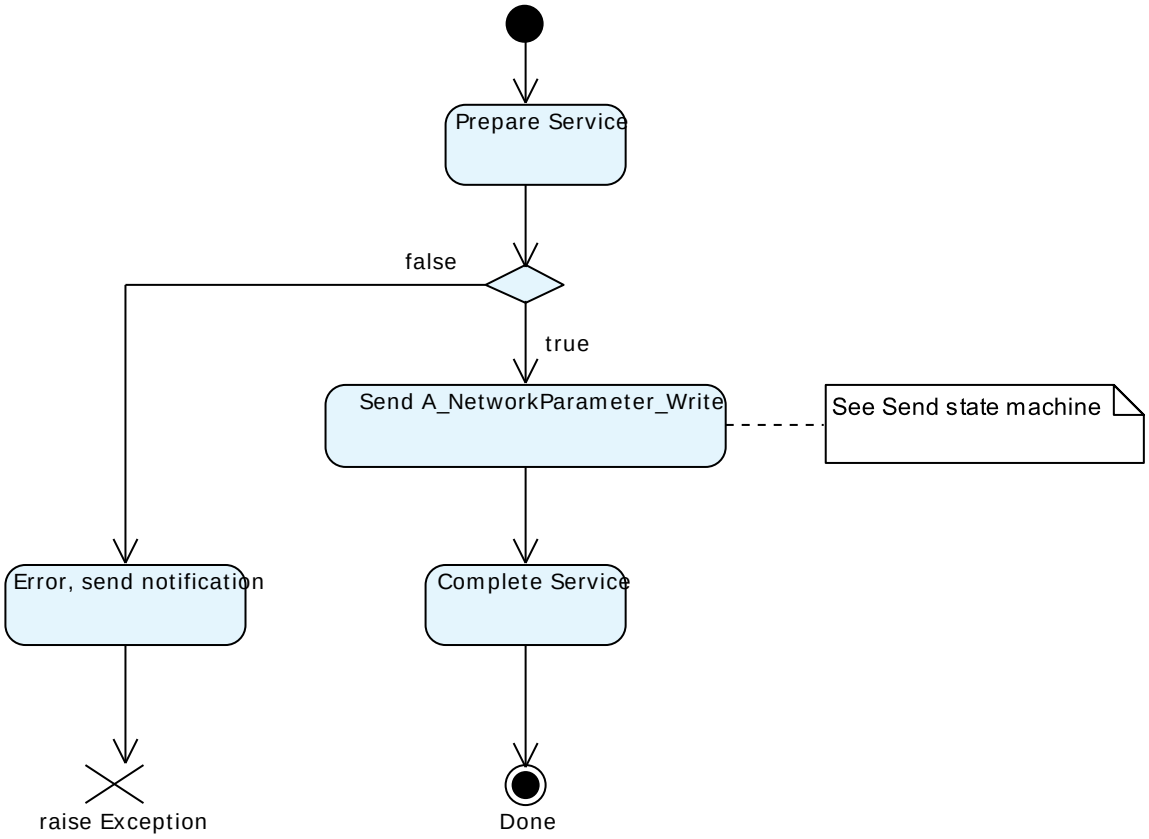


Finite State Machine: IndividualAddressSerialNumberWrite

3.5.6. NetworkParameterWriteBroadcast

Table 3.38. NetworkParameterWriteBroadcast

Attribute	Description
Name	NetworkParameterWriteBroadcast
Description	Writes an NetworkParameter to all devices in the network.
Broadcast Type	System Broadcast. Note: Some Network Management Procedures (e.g. PID_LOCALISATION_MODE) require that this service is sent as Broadcast and not SystemBroadcast. To do this, set the property setSystemBroadcast(false) before running this service.
Inputs	ObjectType (integer) [0 .. 0xFFFF] Property ID (integer) [1 .. 0xFF] Value (byte array) [1 .. 11 bytes]
Outputs	No Outputs
Validation	None
Specification	03_03_07 Application Layer 3.2.7 A_NetworkParameter_Write-service

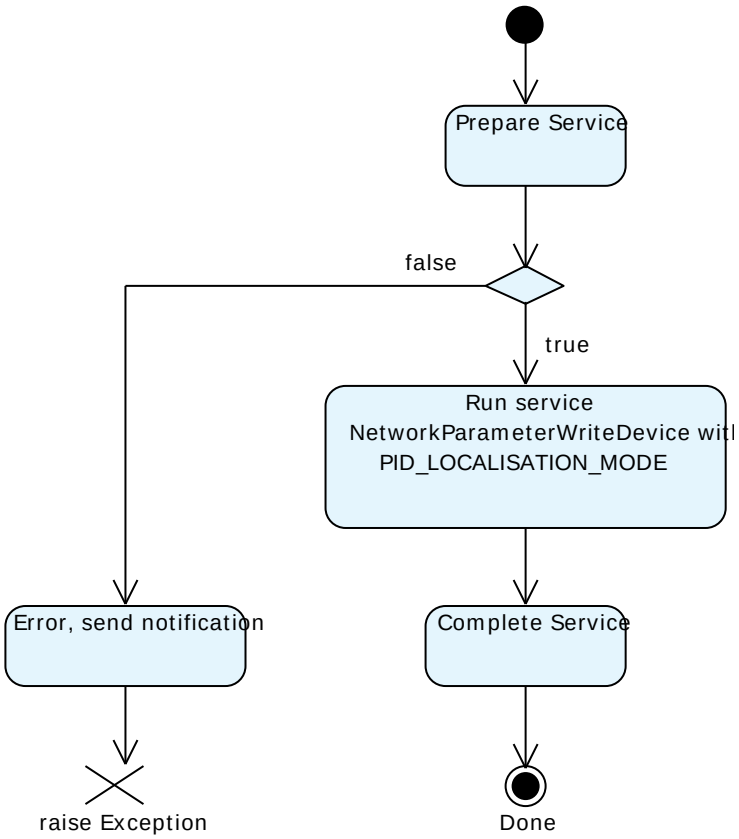


Finite State Machine: NetworkParameterWriteBroadcast

3.5.7. WriteLocalisationMode

Table 3.39. WriteLocalisationMode

Attribute	Description
Name	WriteLocalisationMode
Description	Enables or disables the localization mode for all devices in the installation. If localization is enabled then the device may send LocalizationReports. This service uses *Section 3.5.6, "NetworkParameterWriteBroadcast"* with Object Type = E-Mode Device Object and PID_LOCALISATION_MODE. Note: After 3 min 30 s after last activation, the devices autonomously inactivate the localisation mode. See also *Section 3.7.1, "LocalisationReport"*
Broadcast Type	Broadcast
Inputs	Enable (bool)
Outputs	No Outputs
Validation	None
Specification	AN134 v02 Flexible E-Mode Channels DV 2.3.7.4 E-Mode Device Object – PID_LOCALISATION_MODE



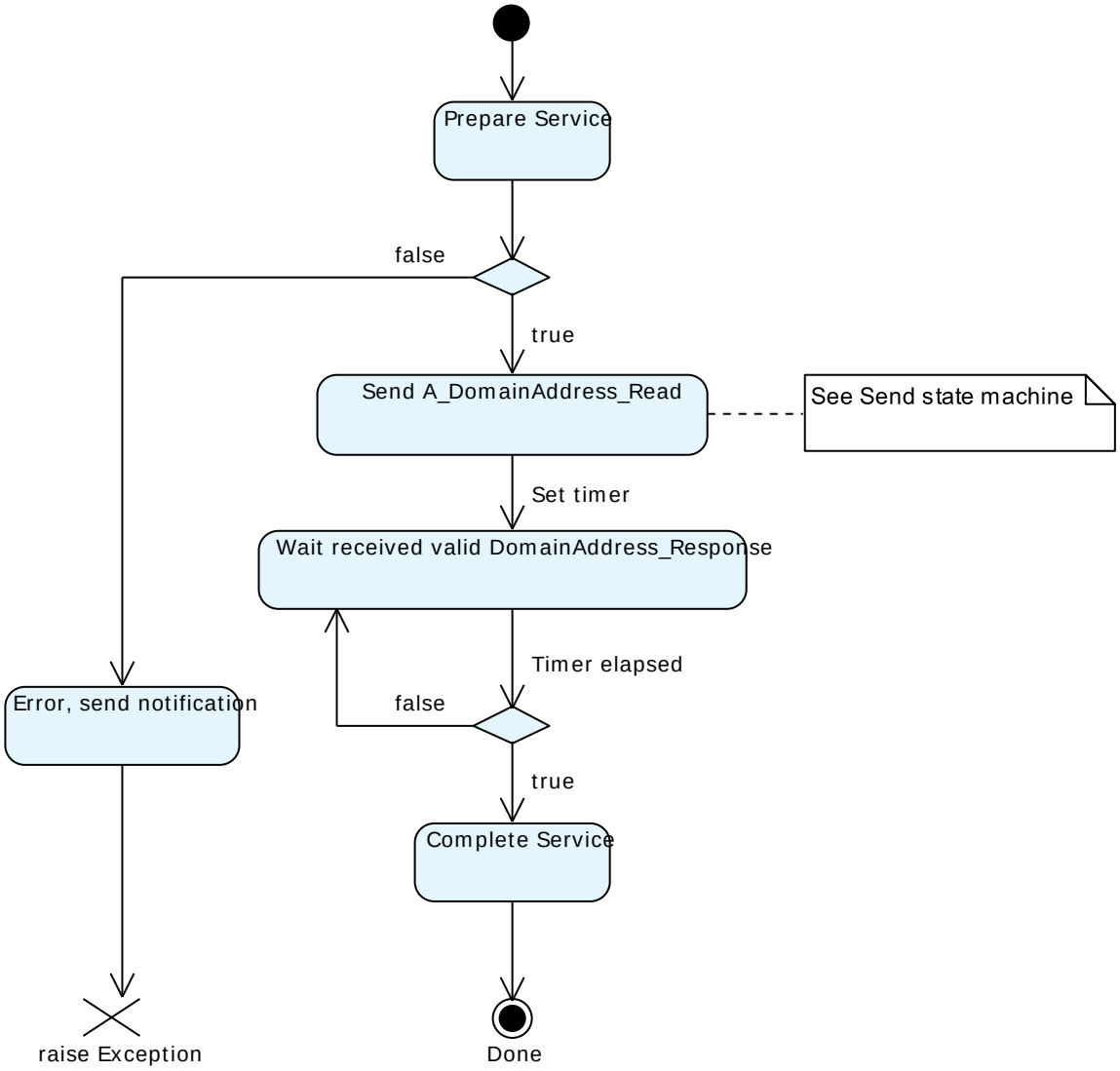
Finite State Machine: WriteLocalisationMode

3.5.8. DomainAddressProgModeRead

Table 3.40. DomainAddressProgModeRead

Attribute	Description
Name	DomainAddressProgModeRead

Attribute	Description
Description	Reads the Domain Addresses of devices in Programming Mode (i.e. with the Learn LED enabled)
Broadcast Type	System Broadcast
Inputs	Wait time in ms (integer, optional default = 500ms)
Outputs	SerialNumbers (list of byte array) Individual Addresses (list of integer) Domain Addresses (list of byte array)
Validation	None
Specification	03_03_07 Application Layer 3.3.4 A_DomainAddress_Read-service



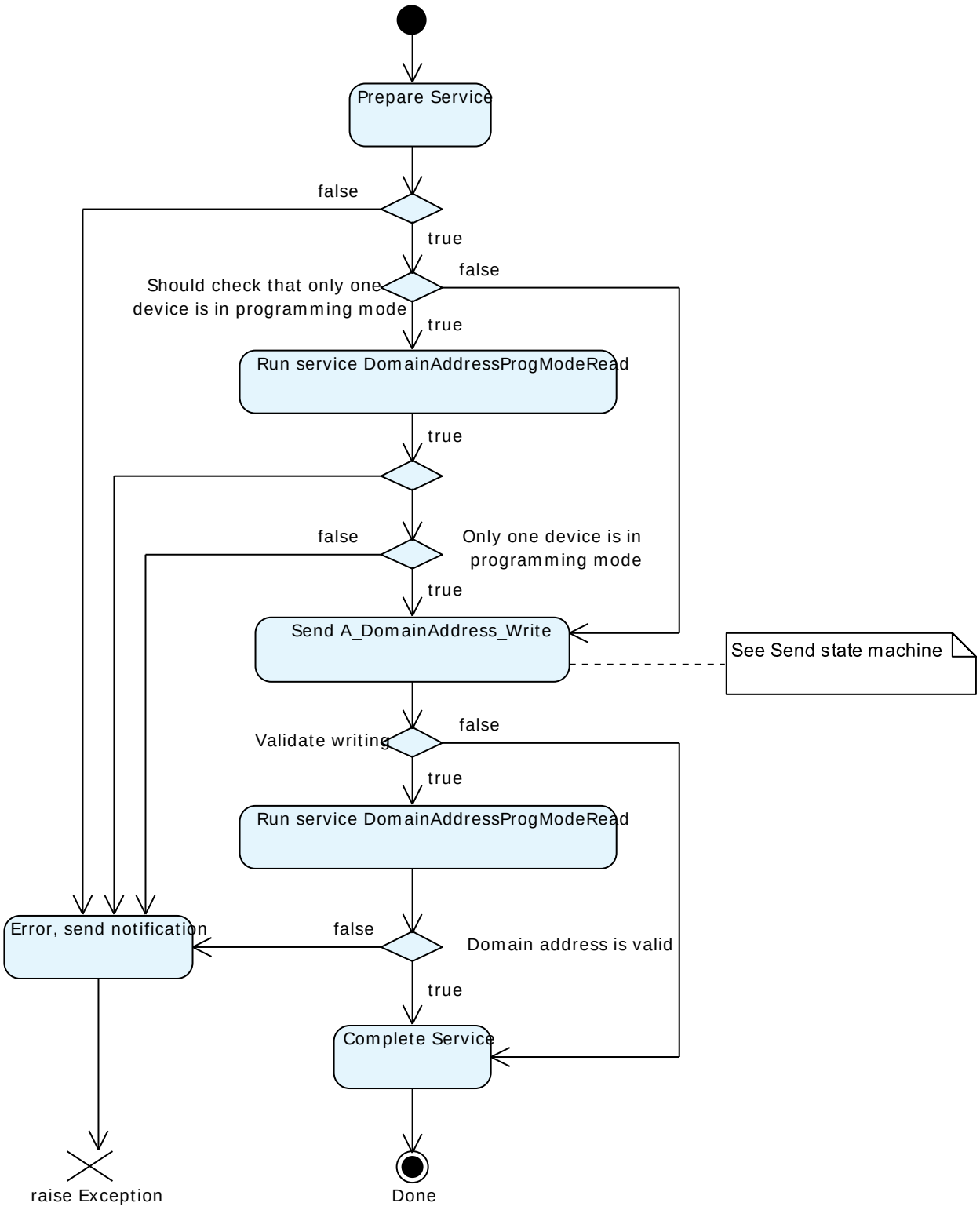
Finite State Machine: DomainAddressProgModeRead

3.5.9. DomainAddressProgModeWrite

Table 3.41. DomainAddressProgModeWrite

Attribute	Description
Name	DomainAddressProgModeWrite

Attribute	Description
Description	Writes the Domain Address to a device in Programming Mode (i.e. with the Learn LED enabled)
Broadcast Type	System Broadcast
Inputs	New Domain Address (byte array) [6 bytes] Check only one device is in ProgMode (Boolean, optional default = true) Validate Writing (Boolean, optional default = true) Wait time for reading in ms (integer, optional default = 500ms)
Outputs	No Outputs
Validation	There is only one device in programming mode (optional) Write verify: the domain address was successfully written (optional)
Exceptions	NoDeviceInProgModeException MoreDevicesInProgModeException WriteDomainAddrFailedException
Specification	03_03_07 Application Layer 3.3.3 A_DomainAddress_Write-service



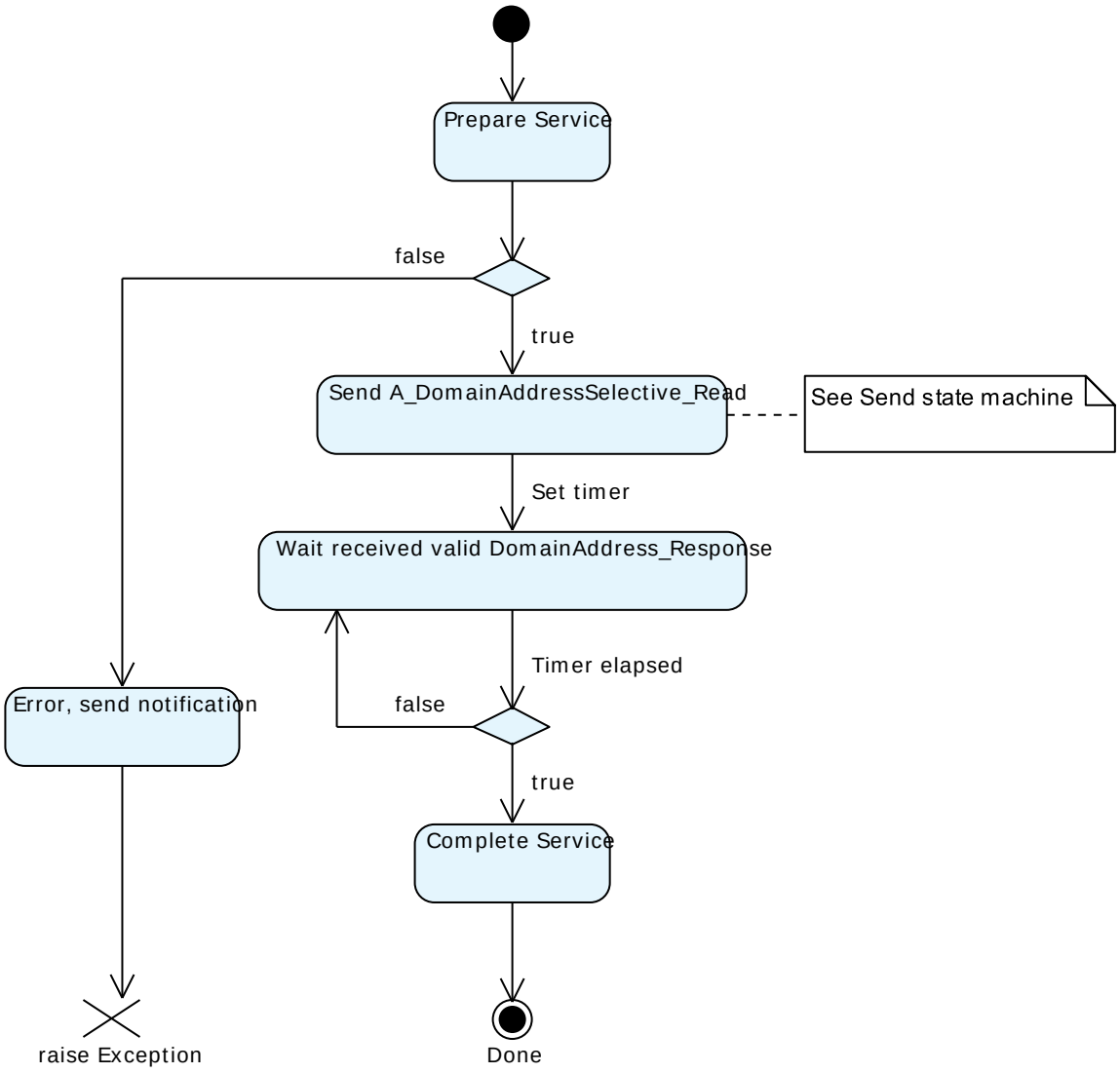
Finite State Machine: DomainAddressProgModeWrite

3.5.10. DomainAddressSelectiveRead

Table 3.42. DomainAddressSelectiveRead

Attribute	Description
Name	DomainAddressSelectiveRead

Attribute	Description
Description	Reads the Domain Address of devices in the specified address range.
Broadcast Type	System Broadcast
Inputs	Type (integer) [1 .. 2] Start Domain Address (byte array) [6 bytes] End Domain Address (byte array) [6 bytes] Wait time for responses in ms (integer, optional default = 2500ms)
Outputs	SerialNumbers (list of byte array) Individual Addresses (list of integer) Domain Addresses (list of byte array)
Validation	None
Specification	AN133 v02 A_DomainAddressSelective_Read DV

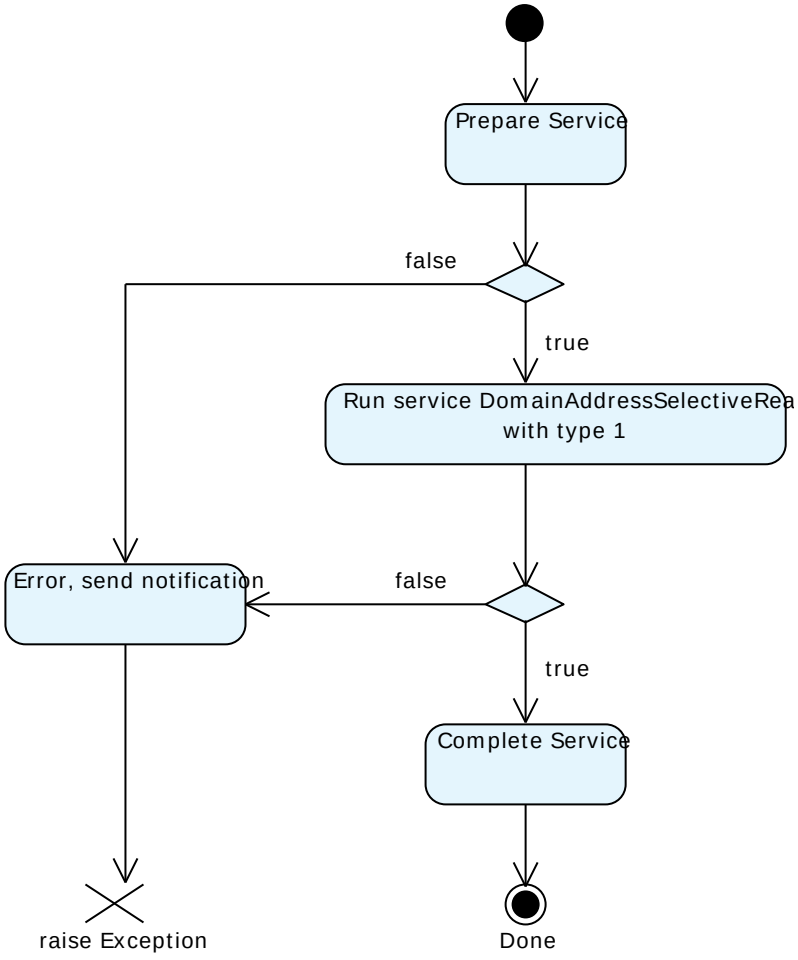


Finite State Machine: DomainAddressSelectiveRead

3.5.11. DomainAddressSelectiveType1Read

Table 3.43. DomainAddressSelectiveType1Read

Attribute	Description
Name	DomainAddressSelectiveType1Read
Description	Reads the Domain Address of all devices in the specified address range. Calls the service *Section 3.5.10, "DomainAddressSelectiveRead"* with type 1.
Broadcast Type	System Broadcast
Inputs	Start Domain Address (byte array) [6 bytes] End Domain Address (byte array) [6 bytes] Wait time for responses in ms (integer, optional default = 2500ms)
Outputs	SerialNumbers (list of byte array) Individual Addresses (list of integer) Domain Addresses (list of byte array)
Validation	None
Specification	AN133 v02 A_DomainAddressSelective_Read DV 2.3.3 Type 01h - six octet DoA page 6

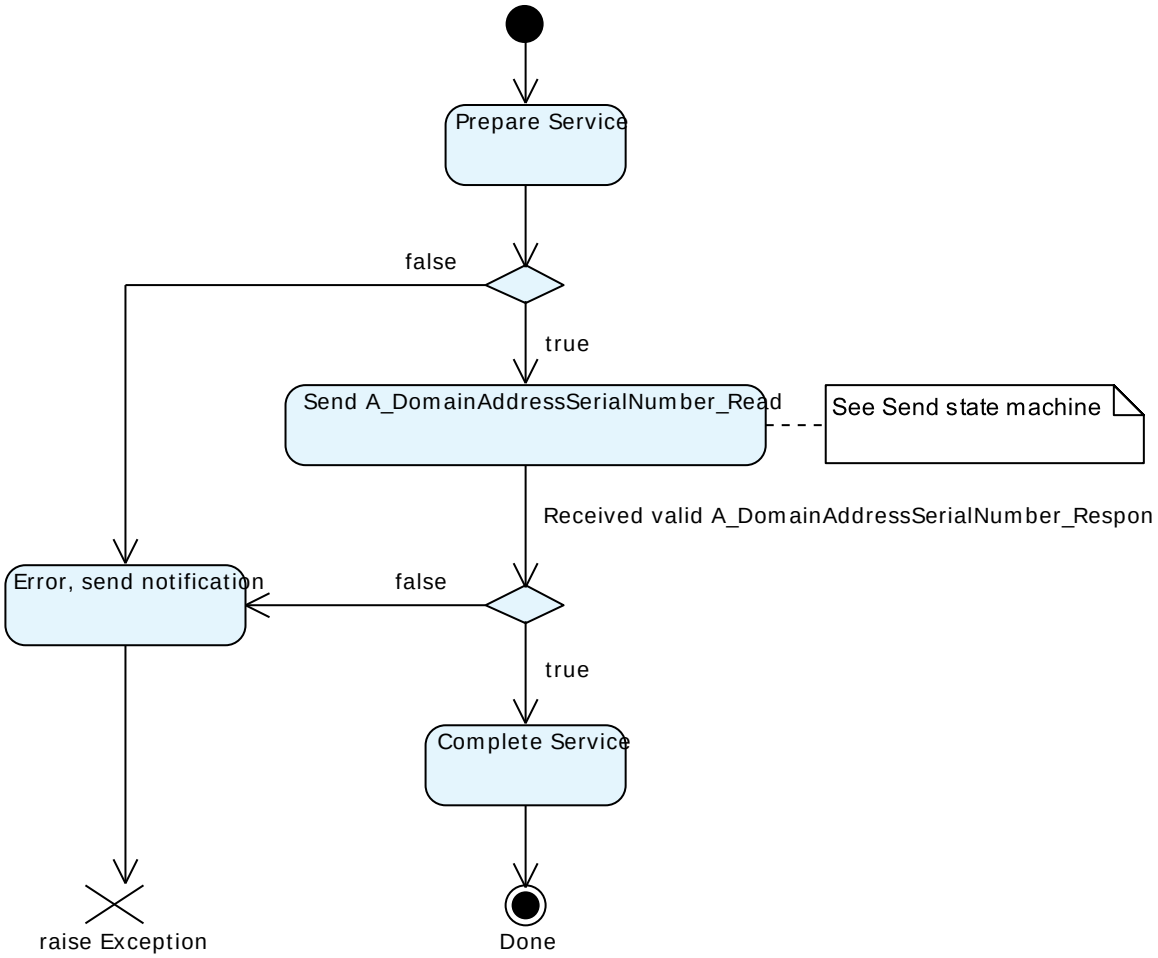


Finite State Machine: DomainAddressSelectiveType1Read

3.5.12. DomainAddressSerialNumberRead

Table 3.44. DomainAddressSerialNumberRead

Attribute	Description
Name	DomainAddressSerialNumberRead
Description	Reads the Domain Address of a device that matches the specified Serial Number
Broadcast Type	System Broadcast
Inputs	Serial Number (6 byte KNX serial number) [6 bytes]
Outputs	Individual Address (integer) Domain Address (byte array)
Validation	None
Exceptions	DeviceErrorException. Thrown when the Serial Number do not match the associated request parameters.
Specification	03_03_07 Application Layer 3.3.6 A_DomainAddressSerialNumber_Read-service

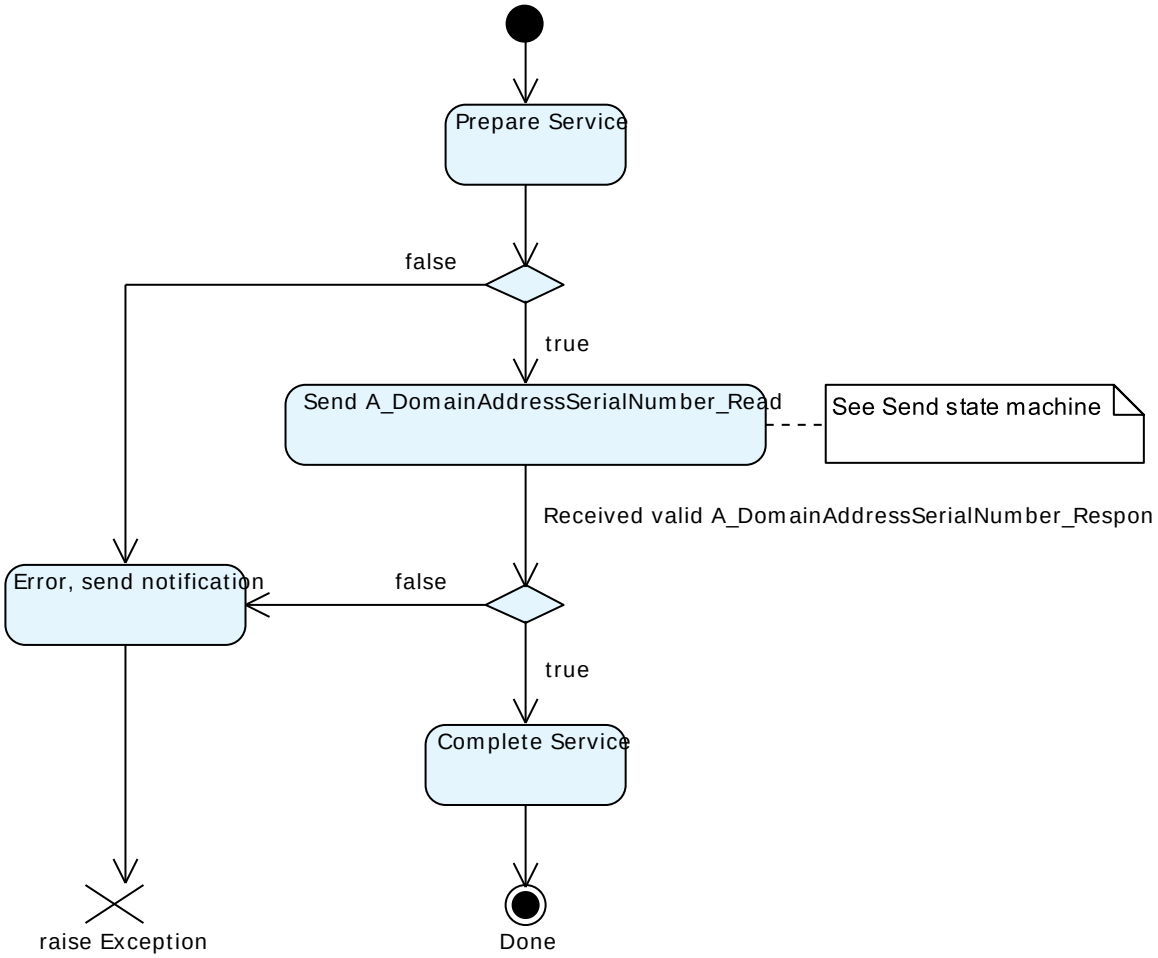


Finite State Machine: DomainAddressSerialNumberRead

3.5.13. DomainAddressSerialNumberWrite

Table 3.45. DomainAddressSerialNumberWrite

Attribute	Description
Name	DomainAddressSerialNumberWrite
Description	Writes the Domain Address of a device
Broadcast Type	System Broadcast
Inputs	Serial Number (byte array) [6 bytes] Domain Address (byte array) [6 bytes] Validate (boolean, optional default = true)
Outputs	No Outputs
Validation	Write verify: the domain address was successfully written (option-al)
Exceptions	WriteDomainAddrFailedException Exceptions from service DomainAddressSerialNumberRead.
Specification	03_03_07 Application Layer 3.3.7 A_DomainAddressSerialNumber_Write-service

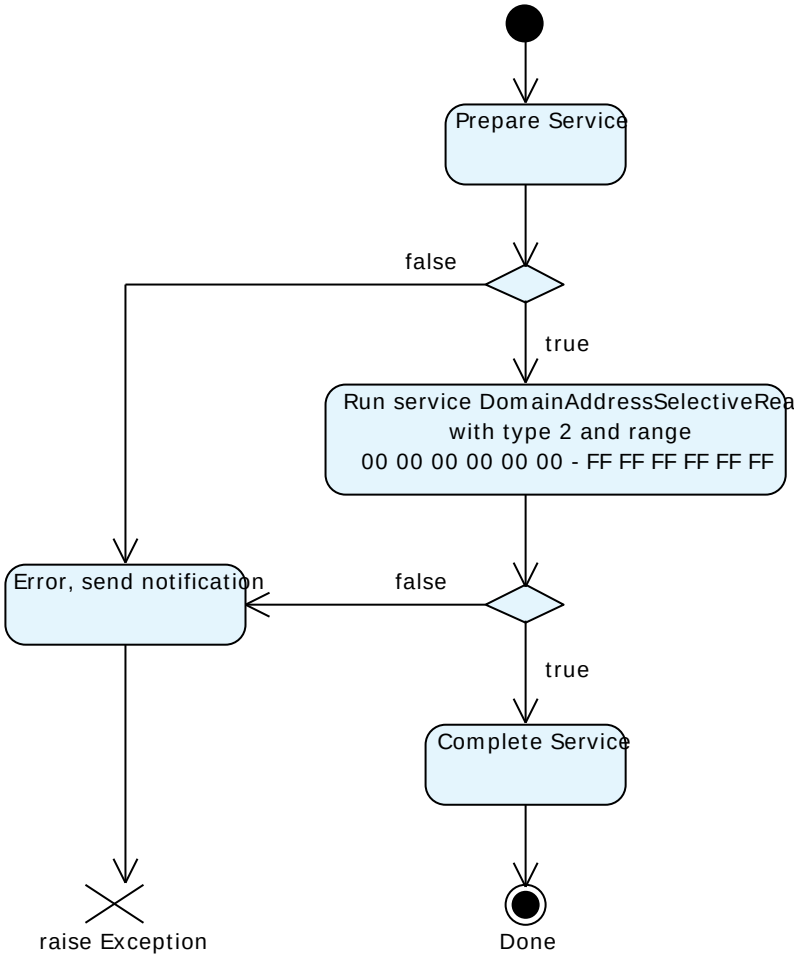


Finite State Machine: DomainAddressSerialNumberWrite

3.5.14. FindJustPoweredDevices

Table 3.46. FindJustPoweredDevices

Attribute	Description
Name	FindJustPoweredDevices
Description	Search just powered devices. Calls the service *Section 3.5.10, "DomainAddressSelectiveRead"* with type 2, start domain address 00 00 00 00 00 00 and end domain address FF FF FF FF FF FF.
Broadcast Type	System Broadcast
Inputs	Wait time for responses in ms (integer, optional default = 2500ms)
Outputs	SerialNumbers (list of byte array) Individual Addresses (list of integer) Domain Addresses (list of byte array)
Validation	None
Specification	Currently not in specification



Finite State Machine: FindJustPoweredDevices

3.6. Group Services

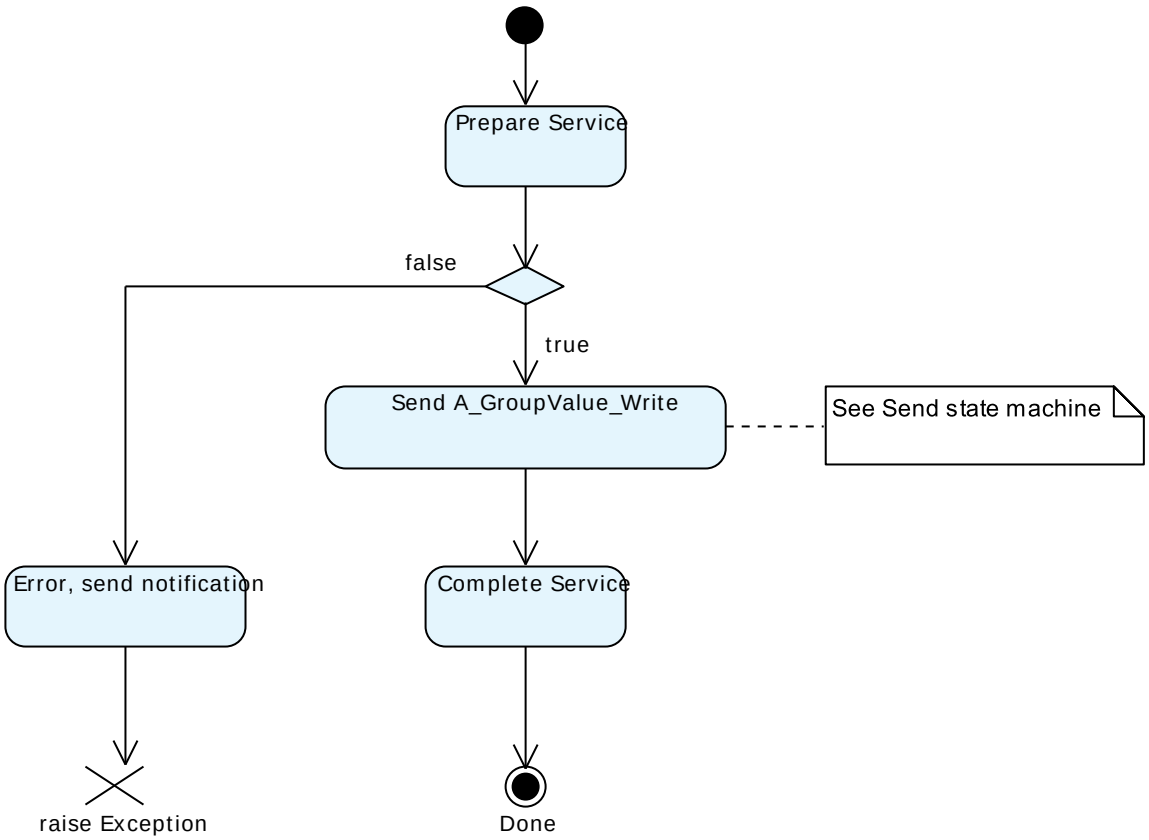
The KNX runtime telegrams, A_GroupValue_Write, A_GroupValue_Read and A_GroupValue_Response are the basis for the kdrive Group Services. There are currently two ser-

vices: GroupValueWrite (which sends a A_GroupValue_Write telegram) and GroupValueRead (which sends a A_GroupValue_Read telegram and waits for a A_GroupValue_Response telegram).

3.6.1. GroupValueWrite

Table 3.47. GroupValueWrite

Attribute	Description
Name	GroupValueWrite
Description	Sends a A_GroupValue_Write telegram
Inputs	GroupAddress (integer) the telegram Group Address (destination) attribute Data (byte array) [1 .. 14 bytes] Group Address Data SizeInBits (integer) the size in bits if the data length is less than or equal to 6 bits SystemBroadcast (boolean) Set to send as SystemBroadcast, default is Broadcast
Outputs	No outputs
Validation	None
Exceptions	InvalidInputParameterException. Thrown when the data is empty, when sizeInBits is set and is out of the range of 1..6 or when sizeInBits is set, is within the range of 1..6 and the data length is greater than 1 byte.
Specification	03_03_07 Application Layer 3.1.3 A_GroupValue_Write-service

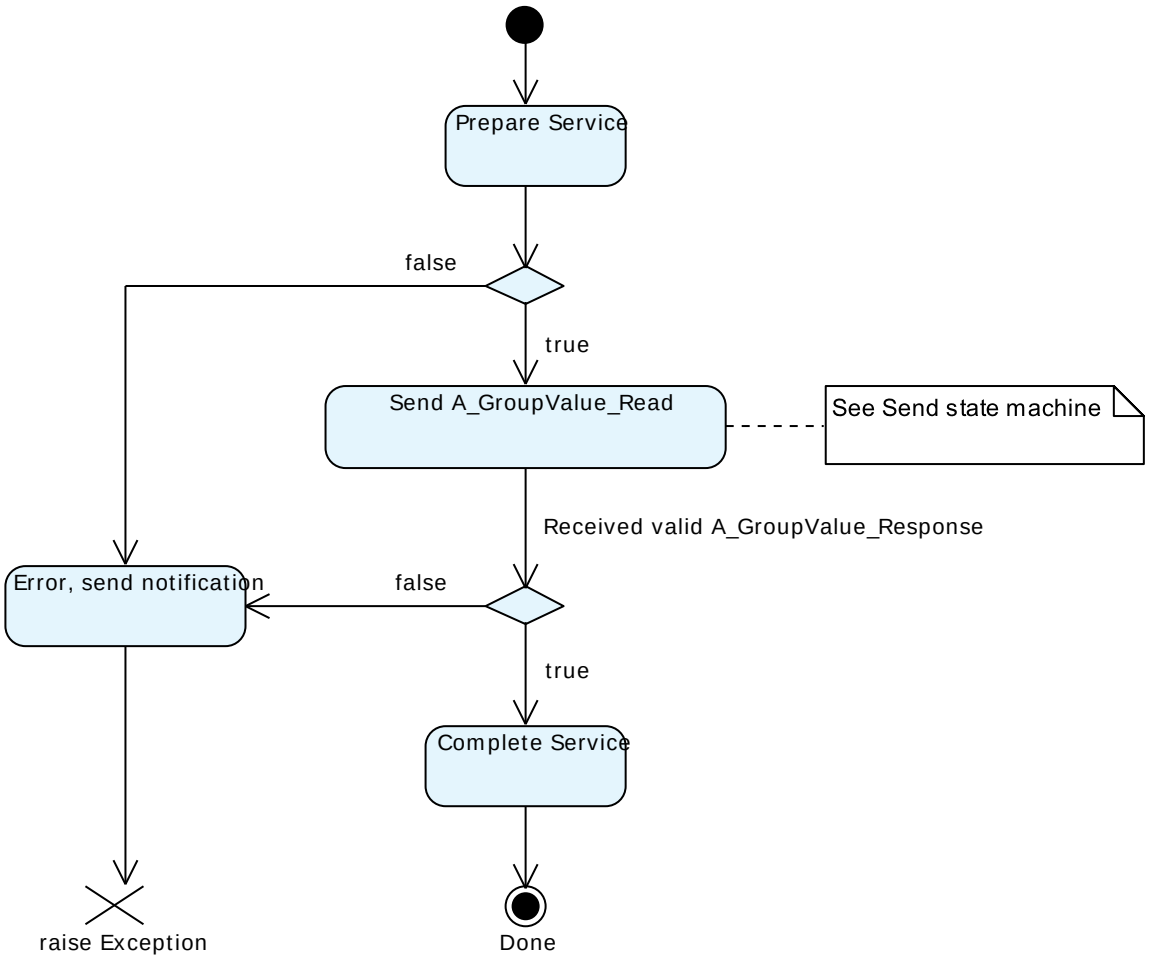


Finite State Machine: GroupValueWrite

3.6.2. GroupValueRead

Table 3.48. GroupValueRead

Attribute	Description
Name	GroupValueRead
Description	Reads a datapoint value from the Bus with the A_GroupValue_Read and A_GroupValue_Response telegrams
Inputs	GroupAddress (integer) the telegram Group Address (destination) attribute SystemBroadcast (boolean) Set to send as SystemBroadcast, default is Broadcast Wait time in ms (integer, optional default = 500ms)
Outputs	Data (byte array) [1 ..14 bytes] Group Address Data
Validation	Via A_GroupValue_Response
Exceptions	See *Section 3.3, "Exceptions"*
Specification	03_03_07 Application Layer 3.1.2 A_GroupValue_Read-service



Finite State Machine: GroupValueRead

3.7. Reports

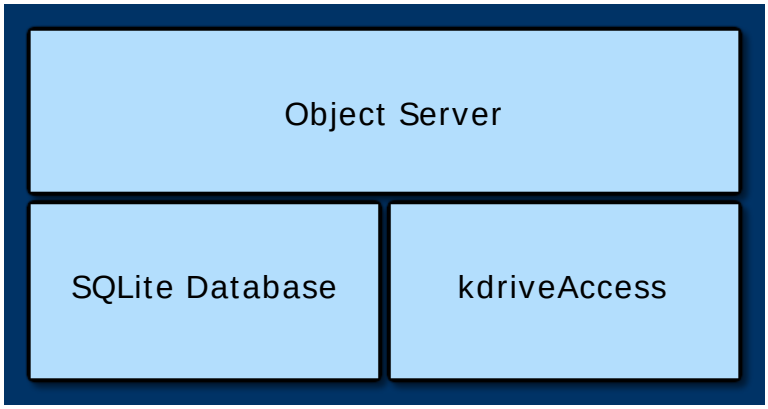
Reports are not directly services but provide service-like information by utilizing one or more management services defined above.

3.7.1. LocalisationReport

The LocalisationReport offers a simple mechanism for receiving Localisation Reports (Individual Address and Channel Number) by sending the WriteLocalisationMode service and waiting for the responses (A_NetworkParameter_InfoReport). As the WriteLocalisationMode has a default timeout period (on the bus) it is periodically sent (until disconnect is called). Each time a valid response is received the Individual Address and Channel Number are extracted and a user notification callback is called.

Object Server

The Object Server is a data respository for KNX Datapoint Values. Internally the Object Server works with an Access Port (from kdriveAccess) and filters for KNX Group Telegrams (GroupValue_Read, GroupValue_Write and GroupValue_Response). It provides an association mapping between Group Addresses and Datapoints (Communication Objects, also known as Group Objects). When a Group Telegram arrives with a Group Address (destination address) the Object Server checks its Association Table for a connection between the address and Communication Objects. Once an association (0 or more) is found the corresponding Communication Objects values are updated.



4.1. Object Store

The Object Server stores its data in the Object Store. The Object Store is an interface to a backend data store, which is responsible for holding the Communication Objects, the Group Addresses and the associations. There are two Object Stores currently available: one based on a SQLite database and the other on STL maps (and multimaps). It is possible to add a custom backend by implementing the Object Store interface.

4.2. Group Object Associations

The Object Server works with Group Object Associations, which define a mapping between a Group Address and a Group Object. This relationship can be many to many: multiple group addresses can be associated with a Group Object (one Send Address, the others Receive Addresses) and multiple Group Objects can be associated with a single Group Address. The associations implemented by the Object Server are modeled on the Association Table of the ETS Managed KNX Stack and offer the same functionality. Each association has a send address (mandatory) and 0 or more optional receive addresses. Note: the send address is also a receive address, that is, it will trigger an event if a Group Value telegram is received from the bus on the send address (assuming the communication object flags are set with C | W). When setting the associations the Object Store requires a send address id and a vector of receive address id's. The receive vector can be empty. Internally the send address gets added to the receive associations, so when you call the Object Store function getComObjectReceiveAssociation you get both the send address and the receive addresses. When you call getAssociations(...) however the receive address vector will not contain the send address. When removing a Group Address from the Object Store it will check to see if an association has been created. It is not possible to remove a Group Address that is part of an association. It is possible to remove a ComObject with associations however, as both the ComObject and its respective Associations will be removed. Note: for RF even though the send address is also used as a receive address it should not actually trigger an event. This is because the send address should be configured with the Serial Number of the sending device (i.e. an RF extended Group Address). For RF if you only set the Send address association (and not one or more optional receive addresses) the ComObject will never be updated

from the bus. For Twisted Pair this is not the case (a Send Address can be used to both send and receive from the bus).

4.3. Datapoints: Communication Objects

An Object Server datapoint is implemented as the ComObject class. This is modeled on the ETS4 Communication Object and has the following mandatory attributes. Note Communication Flags values are: 1 for Enabled, 0 for Disabled.

Table 4.1. Mandatory Communication Object Attributes

Attributes	Description
Id	A unique ID representing the Communication Object. Does not contain any semantic information.
Name	The Communication Object Name.
Number	A unique Communication Object Number. This number is used in the Object Server to identify the ComObject.
ObjectSize	The ObjectSize represents the Datapoint Size, and can be one of the following: 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 Bit 7 Bit 1 Byte 2 Bytes 3 Bytes 4 Bytes 5 Bytes 6 Bytes 7 Bytes 8 Bytes 9 Bytes 10 Bytes 11 Bytes 12 Bytes 14 Bytes LegacyVarData See kdrive::knx::ObjectSize.
ReadFlag	The value of this object can be read via the bus if this flag is set.
WriteFlag	The value of the object can be modified via the bus if this flag is set.
CommunicationFlag	The communication object has a link to the bus if this flag is set.
TransmitFlag	A telegram will be transmitted when the object value (in the object store) has been modified if this flag is set.

Attributes	Description
UpdateFlag	Value response telegrams are interpreted as a write command and hence the value of the group object will be updated if this flag is set.
ReadOnInitFlag	The object reads the value from bus during initialization of the object server.

The following attributes are optional:

Table 4.2. Mandatory Communication Object Attributes

Attributes	Description
Text	Text for visualisation etc. Application specific. Not used by the Object Server.
FunctionText	Text for visualisation etc. Application specific. Not used by the Object Server.
VisibleDescription	Text for visualisation etc. Application specific. Not used by the Object Server.
Priority	one of: Low, Normal, Urgent and System. See kdrive::knx::ControlField.
DatapointType	Text representing the Datapoint Type or Datapoint Subtype. i.e. DPT1. Application specific, not used by the Object Server.
Value	The Datapoint Value. This is a byte array (vector) of up to 14 bytes and is validated against the ObjectSize. This can be empty.
LastUpdate	The timestamp of when the data was last updated. The timestamp is a 64-bit number expressed in time_t. time_t base time is midnight, January 1, 1970. The resolution is one second.

4.4. Group Address

The Group Address is either a standard Group Address (for example: 1/1/2) or it is coupled with a Serial Number (Extended Group Address). Serial Numbers are only used for open media (RF). If the Group Address is extended (i.e. Group Address + Serial Number) the Object Server inspects the received telegrams to extract the Serial Number from the additional info field. If the Serial Number is not present the telegram is discarded. If it is found the two Serial Numbers are compared, and if they do not match the telegram is discarded. For standard addressing (Group Address without a Serial Number) all telegrams that match the Group Address are accepted.

4.5. Configuration

The Object Server can be configured via API (via the Object Server functions: addComObject, addGroupAddress and setAssociations) and via XML. The XML is described in the Object Server XSD Schema.

4.6. Event Notifications

The ObjectServer implements an Event Notification Signal to notify the user of the following: ComObject Add, Remove, Association Modified, ComObject Data Modified, GroupValue Telegram sent or re-

ceived (and the associated ComObjects). The notification signal is called from the current context of the ObjectServer and not from within its own thread. That means, for all signals invoked from a user call it will be in the user thread context and from all signals resulting from a received telegram the signal will be in the context of the AccessPort Packet Notification thread.

4.7. ComObject Data

Two set functions and two get functions are implemented. The **`setData`** and **`getData`** functions simply set/get the value of the associated ComObject. To send a GroupValue_Write/GroupValue_Read telegram on the bus you use **`writeToBus`** and **`readFromBus`** respectively. Note: **`writeToBus`** also calls **`setData`** and **`readFromBus`** calls **`getData`**. **`writeToBus`** and **`readFromBus`** check the communication flags of the ComObject and only send a telegram if the appropriate flags are enabled. If the flags are not enabled the telegram is not sent. **`writeToBus`** additionally uses a flag to determine whether the telegram should be sent or not and the default is to send only when the value of the com object has been modified. To send the same value multiple times set **`writeRegardless`** to true.

4.8. Object Server Usage

The Object Server requires an Object Store to hold the data and and an open Access Port to send and receive from the KNX Bus. See the kdriveAccess section for more information on the Access Port.

Example 4.1. Sample: Creating and configuring the Object Server

```
ObjectStore::Ptr objectStore(new STLObjectStore);
ObjectServer objectServer(objectStore, accessPort);

// Create a Communication Object (C | T | W | U)
// Can write to the bus, and be updated from the bus
ComObject comObject;
comObject.setId("ComObject1");
comObject.setName("Communication Object 1");
comObject.setNumber(1);
comObject.setObjectSize(ObjectSize::ComObjectSize_1_Bit);
comObject.setWriteFlag(1);
comObject.setCommunicationFlag(1);
comObject.setTransmitFlag(1);
comObject.setUpdateFlag(1);
objectServer.addComObject(comObject);

// Add two Group Address, 1/1/1 and 1/1/2 (we add in hex)
objectServer.addGroupAddress("GA1", 0x901, "Group Address 1");
objectServer.addGroupAddress("GA2", 0x902, "Group Address 2");
objectServer.addGroupAddress("GA3", 0x903, "Group Address 3");

// Set associations for ComObject 1
// GA1 is the send address.
// GA2 and GA3 are receive addresses.
std::vector<std::string> receive;
receive.push_back("GA2");
receive.push_back("GA3");
objectServer.setAssociations(comObject, "GA1", receive);
```

Example 4.2. Sample: Group Value Write

To send a Group Value Write use the function **`writeToBus`**.

```
// create data array
std::vector<unsigned char> data;
data.push_back(0x01);

// send GroupValue_Write
objectServer.writeToBus(1, data);
```

Example 4.3. Sample: Group Value Read

To send a Group Value Read use the function **`readFromBus`**.

```
// Returns the updated ComObject after the GroupValue_Response is received
// If the response is not received we still get the comObject
// (i.e. the same as calling getData)
ComObject::Ptr comObject = objectServer.readFromBus(1);
```

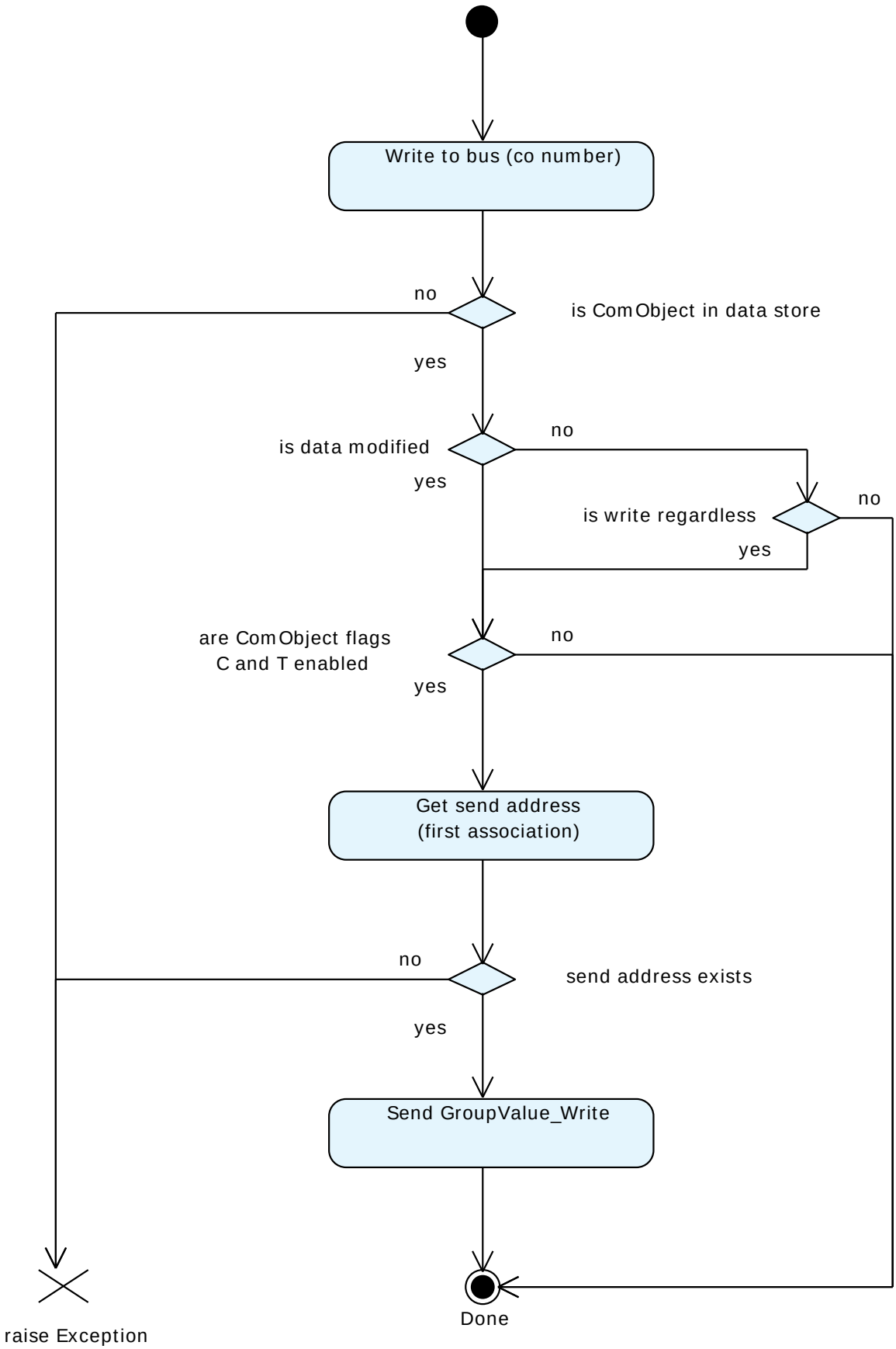
Example 4.4. Sample: Event Notifications

To connect to the Event Notifications use the standard boost:signals2 notification system.

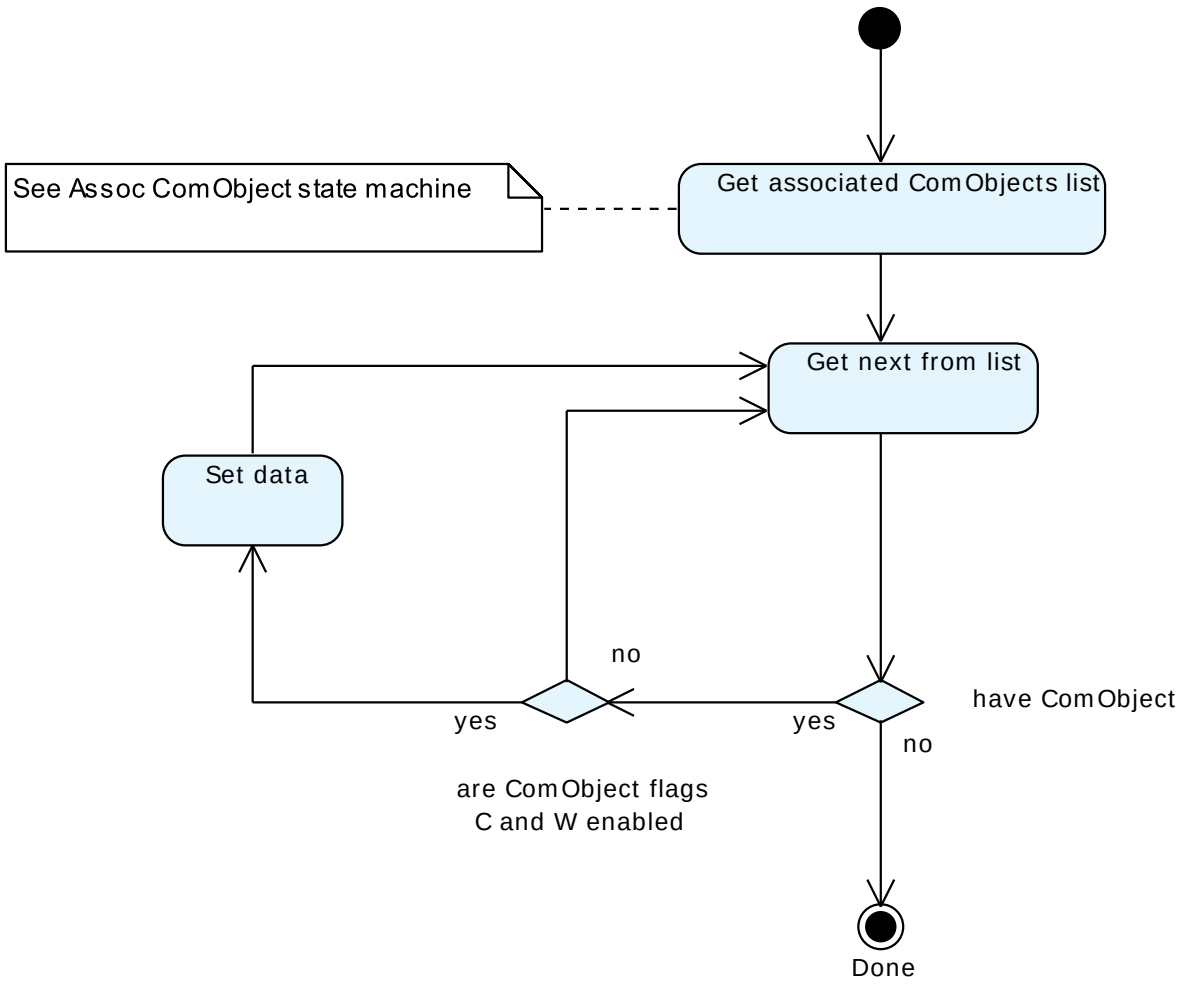
```
// get the event notification signal
ObjectServer::EventSignal& eventSignal = objectServer.getEventSignal();

// connect to the event signal, use a scoped connection to ensure
// we disconnect from the signal when we go out of scope
boost::signals2::scoped_connection connection = eventSignal.connect(onEvent);

...

/*!
  Called when a ObjectServer event is signaled.
  We simply write the ComObjects information (0 or more) to the logger
*/
void onEvent(const ObjectServer::ComObjects& comObjects, int eventId)
{
 poco_information(logger(), ObjectServerEvents::toString(comObjects, eventId));
}
```

4.9. Object Server Finite State Machines

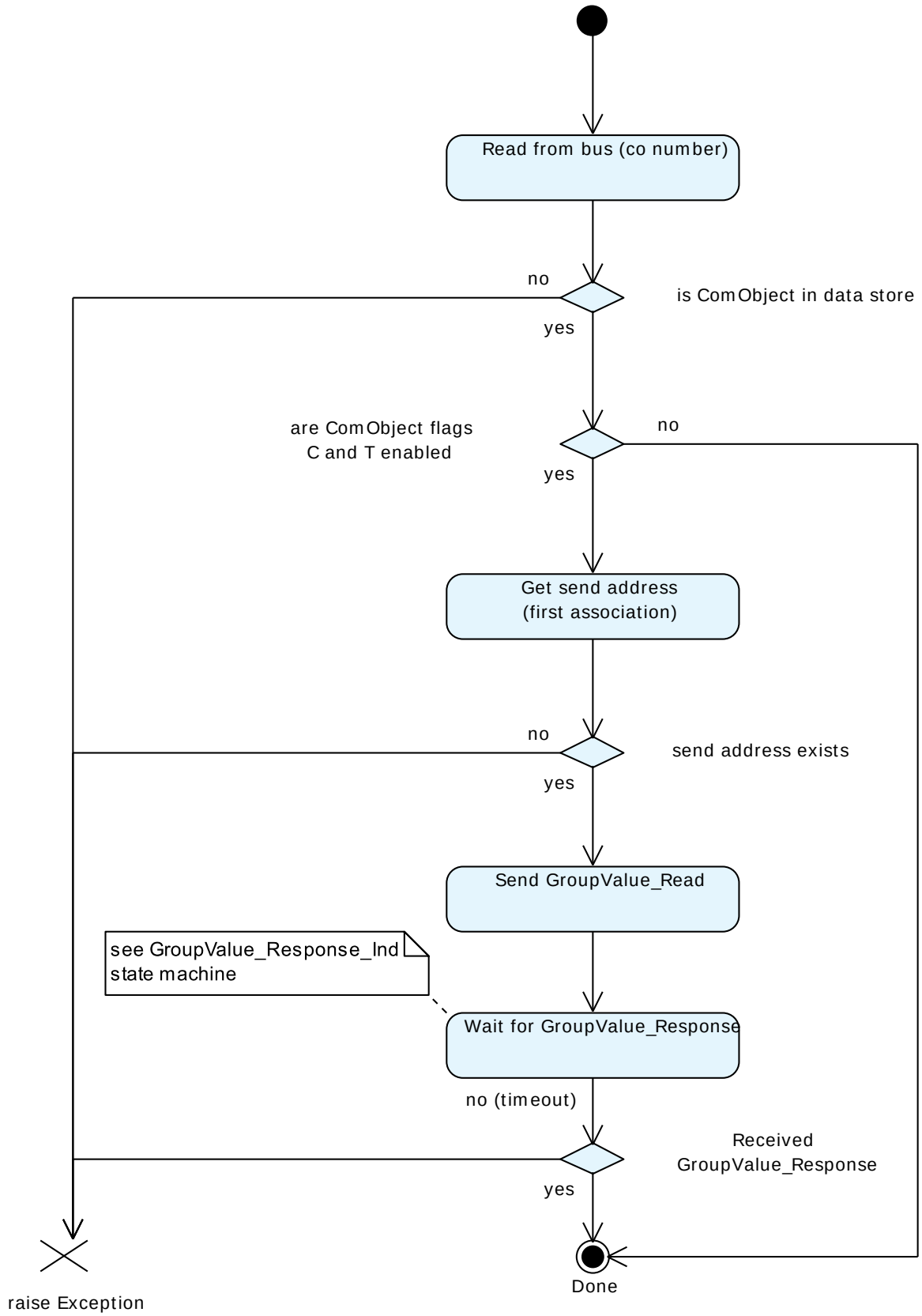
The following state machines describe the process for handling Group Value requests and indications.



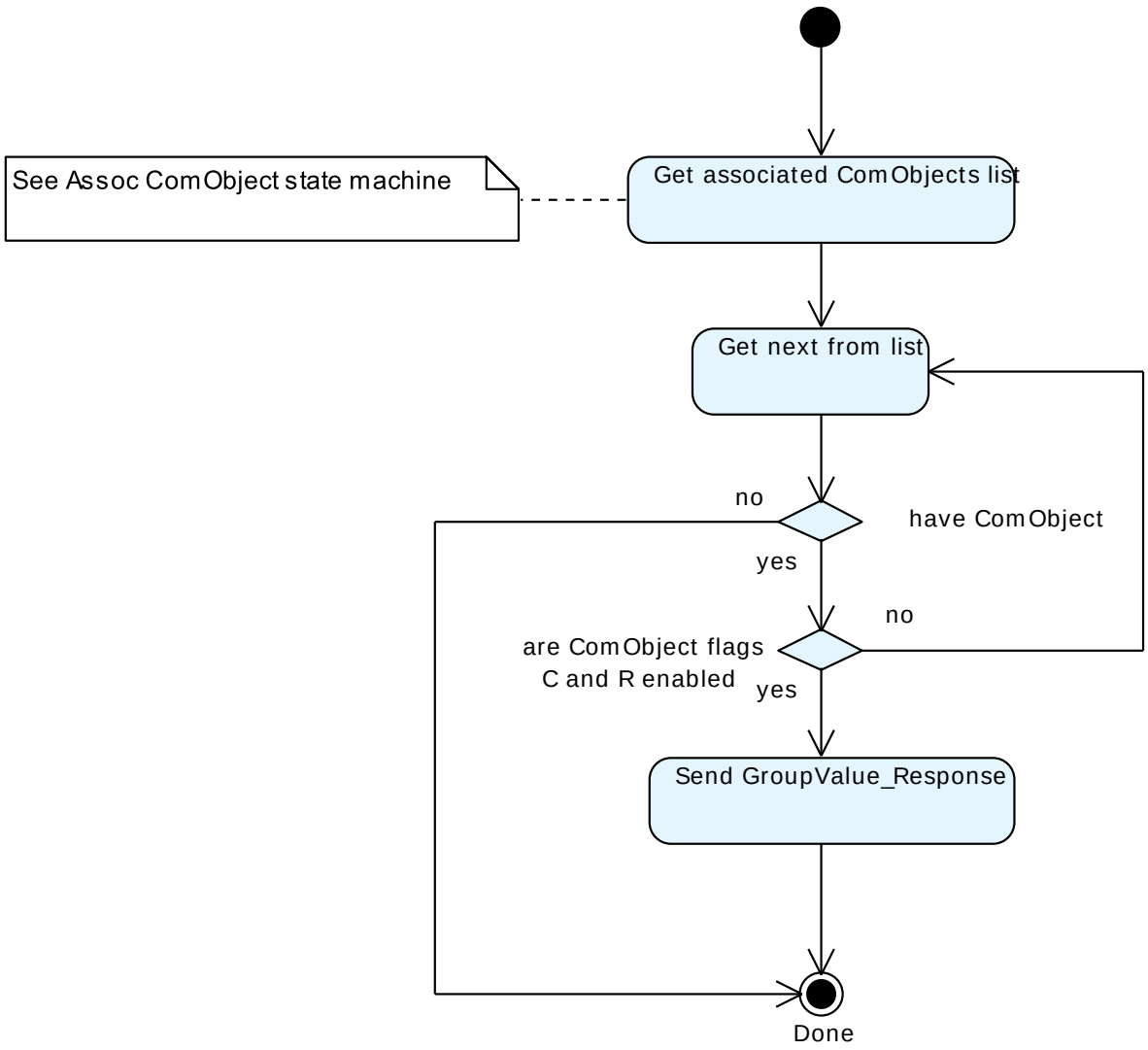
Finite State Machine: Group Value Write Request



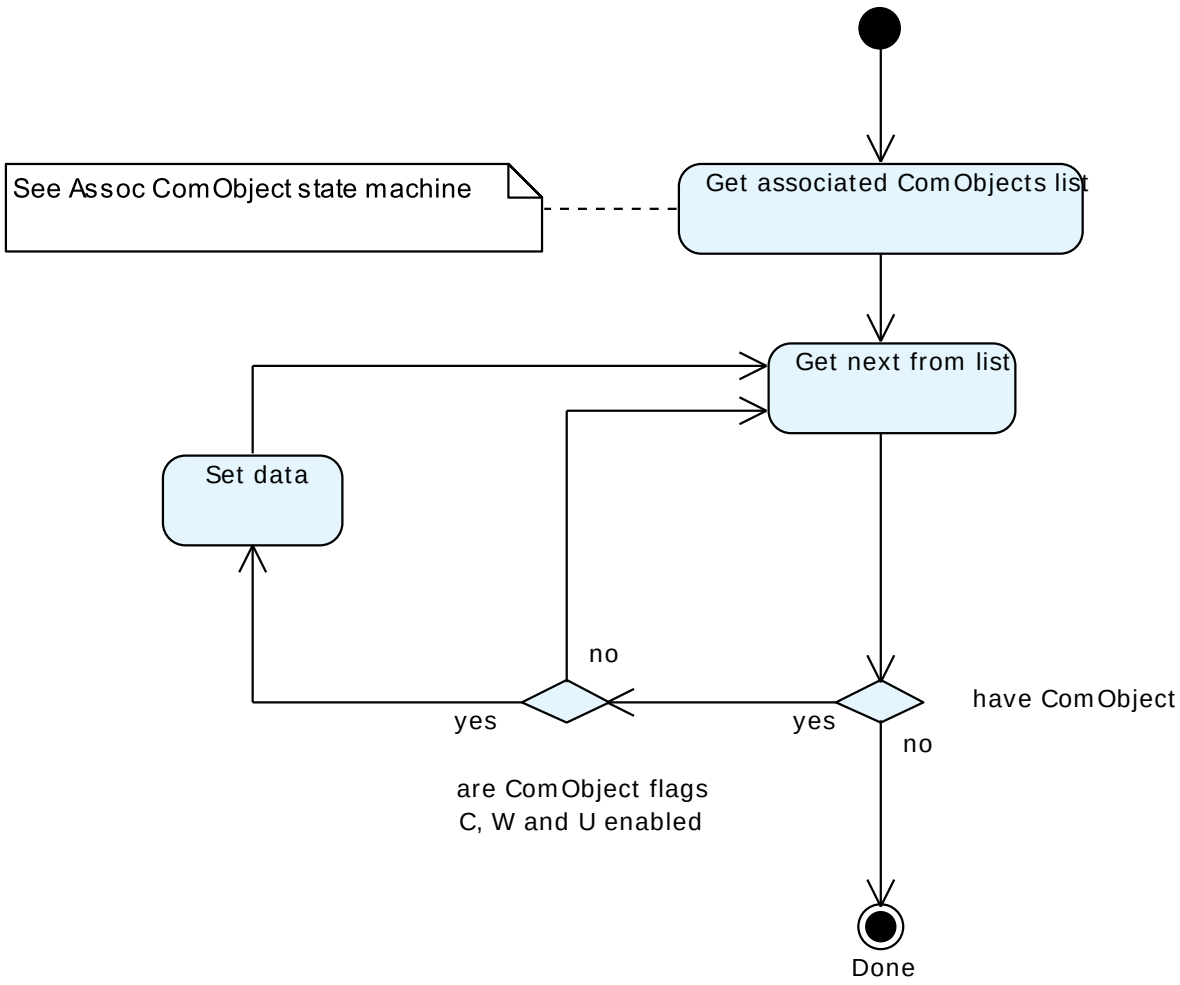
Finite State Machine: Group Value Write Indication



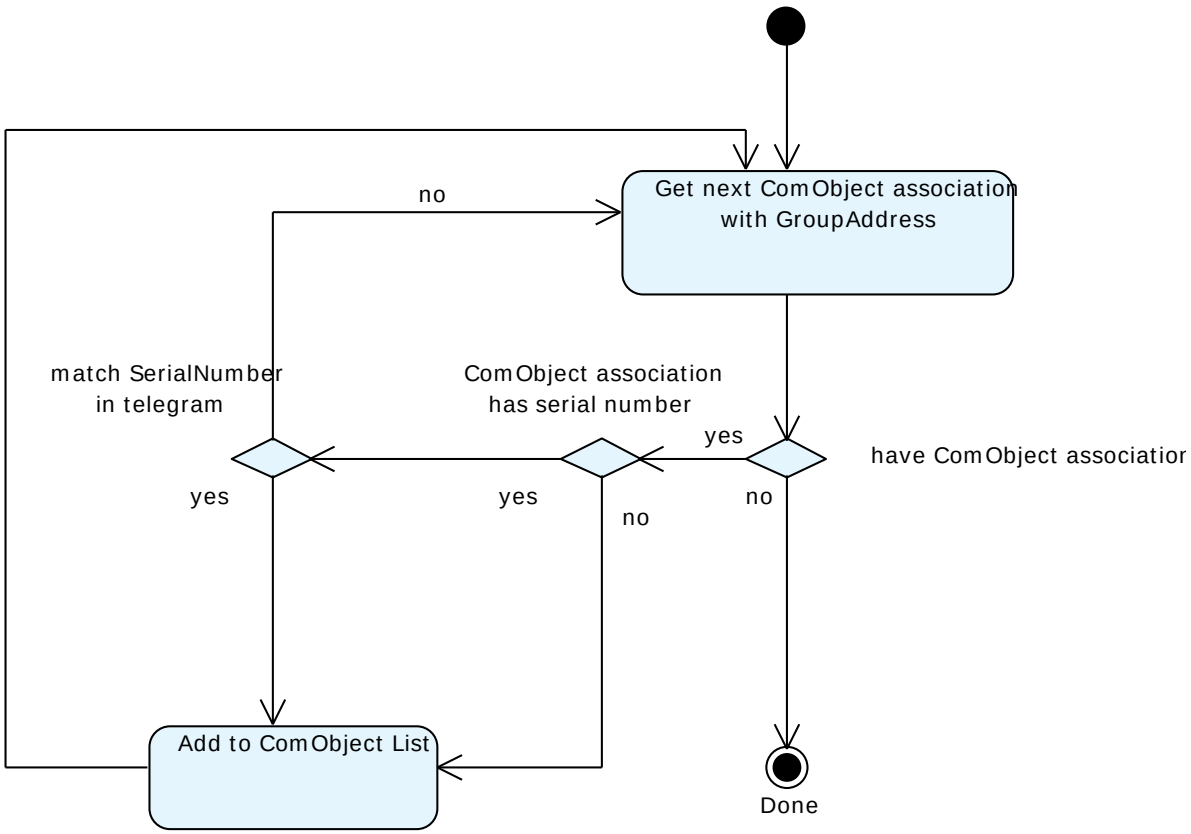
Finite State Machine: Group Value Read Request



Finite State Machine: Group Value Read Indication



Finite State Machine: Group Value Response Indication



Finite State Machine: Get Associated Communication Objects

Kernel

The kdriveKernel library is responsible for the OSI protocol layers from the Link Layer to the Transport Layer. It is specifically used to manage Connection-Oriented communication. The KNX specification details the Transport Layer state machine in the document: 03_03_04 Transport Layer v1.1 AS along with a number of Transition Table styles. kdriveKernel implements Style 3. The Link Layer is provided by the kdriveAccess library.

5.1. Kernel Manager

The Kernel Manager is the main interface to the Protocol Stack. The Kernel Manager is derived from a QueueConnector, which is the same base that all Access Ports derive from so you can use it in much the same way that you use an Access Port. It has a packet signal (for Rx and Tx packets) and implements a recieve queue. Internally it implements an event queue with which it interleaves both the Up (Rx) and Down (Tx) events and the internal timer events from the state machines. The default, when sending a telegram via the Kernel Manager, is for it to add an event to the event queue and return. If you need to wait for the confirm you should use the function **`sendWithFlowControl`**. The Kernel Manager is typically used together with kdriveServices to implement connection-oriented device services, such as MemoryRead. See *Chapter 3, Services* for an example of using the application services with kdriveKernel.

5.2. Exceptions

The kdriveKernal component defines a number of Error Codes (see core/ErrorCodes.h) which also have corresponding Poco::Exception derived Exceptions (i.e. the error codes are propogated throughout the kernel library using Exceptions). Note: The current version of kdriveKernel uses the ErrorCodes defined in core/ErrorCodes.h however is some cases will throw a Poco::Exception with the error code defined and not the corresponding kdriveKernel exception. This will be fixed in a future version.

Table 5.1. Kernel Framework Exceptions

Exception Name	Description
KernelException	kdriveKernel exception. This is the base class for all Kernel Exceptions and is not directly thrown, however can be used to catch all Kernel-based exceptions.
RepeatedException	Internal. Called when a DATA indication is repeated from the remote partner. We do not route this message to the user.
SequenceNumberErrorException	Internal. Called when a DATA indication is received from the remote partner and the sequence numbers are not valid. We do not route this message to the user.
IgnoredException	Internal. Used to indicate that the message should be ignored.
DisconnectedException	Indicates that a connection has been closed, typically by a T_Disconnect primitive. This is raised within **`sendWithFlowControl`** when waiting for a confirm.
RejectedException	With the one shared queue system (in protocol stack) we have to handle requests synchonously for each connection (i.e. we can only send one request at a time for a connection).
NoResponseFromDeviceException	Thrown from within **`sendWithFlowControl`** when waiting for a Confirm. Indicates that a confirm was not received.

Exception Name	Description
NoLayersDefinedException	Raised when starting the Kernel Manager (Protocol Stack) without valid protocol layers. Typically these are added with the call to **`createDefaultLayers`**.
PacketException	This exception is raised when trying to send a Telegram that is not a (or derived from) **`kdrive::access::TelegramPacket`**
NoMoreConnectionsException	The Transport Layer currently limits the number of concurrent connections to 5. This is an abritrary number. When the number and is hardcoded in the library. This exception is raised when there are already max connections active and a new connection is requested.

kdriveExpress

The *kdriveExpress* Library is a "C" interface to (a subset of) the kdrive C++ library. [1] It currently exposes the kdriveAccess and kdriveServices libraries. The kdriveAccess library enables software to communicate with a KNX Interface Devices, such as the KNX USB Interface 310 or the KNX IP Interface 730 from Weinzierl Engineering. The kdriveServices library implements KNX management services, such as PropertyValueRead and PropertyValueWrite for example.

6.1. Getting Started with kdriveExpress C

The *kdriveExpress* Library is available both as a binary Windows DLL and a shared library for Ubuntu Linux, however may also be distributed as a Windows CE DLL or a dynamic library for Mac OS X on request. The C Header Files are found in the includes directory of the SDK, the DLL in the bin directory and the lib files in the corresponding lib directory.

Compiling. The Header files use typedefs for the fundamental types and these can be found in **`kdrive_express_config.h`**. The typedefs are in the form: **`uint32_t`** to represent an unsigned 32 bit integer, for example. If you have these types already defined in your project you can suppress them by adding the precompiler definition: **`KDRIVE_TYPES`**. The error type is defined as **`error_t`** (an int) and can be suppressed by defining: **`KDRIVE_ERROR_TYPE`**. The bool type is defined as **`bool_t`** (an int) and can be suppressed by defining: **`KDRIVE_BOOL_TYPE`**.

Linking. When linking your application for WIN32 you will need to add the following libraries to the linker: **`kdriveExpress.lib`**. When linking for Linux, you need to add: **`-lkdriveExpress -lpthread`**

Running. When running your application under windows you will need to ensure that the library **`kdriveExpress.dll`** is in your path. The simplest solution is to simply copy the dll into the same directory as your executable. See also *Section A.1, "Microsoft Visual C++ Redistributable Package"*

6.1.1. Error Handling

kdriveExpress uses Error Codes to indicate error conditions. Error Codes are defined in **`kdrive_express_error.h`**. The errors are typically translated from an Exception in the C++ Library to a corresponding Error Code in the *kdriveExpress* Library. A function is available to obtain the error message associated with an error code.

6.1.2. Notification Events

There are several notification events available to indicate that an Access Port has been opened, closed or terminated. The event notifications are raised in a separate notification thread (the same thread that also provides telegram notifications, see also *Section 6.1.4.6, "Receiving KNX Telegrams"*.

6.1.3. Logger

The kdrive libraries use the Poco C++ logger framework for logging, and the *kdriveExpress* api enables support for logging either to the console or to a file. It is also possible to write to the logger from your application.

[1] kdrive is a cross-platform C++ software component library from Weinzierl Engineering GmbH that enables the rapid development of KNX applications on platforms with POSIX (like) operating systems, such as Linux, Windows, Windows CE and iOS.

Logger Format Buffer

The **`function kdrive_logger_ex`** implements a sprintf-like interface to the logger with a variable number of arguments. It internally uses a buffer to format the log message. The buffer is static and limited to 1024 bytes. There is a potential for a buffer overflow if you try to write a log message greater than 1024 characters in size. This is the only api function that writes to a buffer without a corresponding value size, and has the potential to overrun the buffer.

6.1.4. Access

The kdrive_access module implements the KNX Interface Bus Access protocols required for communicating with KNX Interface Devices and is used for sending and receiving KNX telegrams, with special emphasis on the runtime communication using GroupValue events.

6.1.4.1. Working with USB

The *kdriveExpress* Library provides support for all USB Interface Devices from Weinzierl Engineering. To open a connection to a USB Interface Device requires a two-step process of i) enumerate and ii) open. The enumeration returns the count of available USB Interface Devices from Weinzierl. The open function takes an enumeration index, which is 0..count - 1. Even if you always open the first available USB device (i.e. index 0) you still have to call USB Enumerate first. As the kdriveAccess library caches the result of the enumeration you only have to call USB Enumerate once during the lifetime of your application.

Example 6.1. Create Connection - USB

The following code sample opens the first USB Connection (usb 0):

```
int32_t ap = kdrive_ap_create();

if (ap == KDRIVE_INVALID_DESCRIPTOR)
{
 application_error_handling_here();
}

if ((kdrive_ap_enum_usb(ap) > 0) && (kdrive_ap_open_usb(ap, 0) == KDRIVE_ERROR_NONE))
{
 ... do something with the access port
}
```

Example 6.2. USB Management under Linux

To access the hid driver under linux as an unprivileged user (i.e. without root permission) you can use udev. The following udev rules simply adds the "knx" group to the device permissions. Accessing the device as non-root simply means adding your user to the configured group (in this case, "knx"). For more information, see the docs/linux/usb directory.

```
            ACTION!="add|change", GOTO="knx_end"
```

```
# [WeinzierlEngineering_0E77_0102]
SUBSYSTEM=="usb", ATTRS{idVendor}=="0e77", ATTRS{idProduct}=="0102", GROUP="knx"

# [WeinzierlEngineering_0E77_0103]
SUBSYSTEM=="usb", ATTRS{idVendor}=="0e77", ATTRS{idProduct}=="0103", GROUP="knx"

# [WeinzierlEngineering_0E77_0104]
SUBSYSTEM=="usb", ATTRS{idVendor}=="0e77", ATTRS{idProduct}=="0104", GROUP="knx"

# [WeinzierlEngineering_0E77_2001]
SUBSYSTEM=="usb", ATTRS{idVendor}=="0e77", ATTRS{idProduct}=="2001", GROUP="knx"

LABEL="knx_end"
```

6.1.4.2. Working with IP Tunneling

Although Enumeration for IP Tunneling is available in the C++ kdrive Library it is not currently supported in the *kdriveExpress* Library. This means you will have to know the IP Address of the Interface Device you wish to communicate with. Two functions are provided to open a connection with an IP Tunneling Device. The first function takes a single IP Address and assume the default network interface adaptor. The second function takes an IP Address and the IP Address of the network interface adaptor. If you have multiple network interface adaptors you may have to use the second function: **`kdrive_ap_open_ip_ex`**.

Example 6.3. Create Connection - IP Tunneling

The following code sample opens a IP Tunneling Connection:

```
int32_t ap = kdrive_ap_create();

if (ap == KDRIVE_INVALID_DESCRIPTOR)
{
 application_error_handling_here();
}

if (kdrive_ap_open_ip(ap, "192.168.1.47") == KDRIVE_ERROR_NONE)
{
 ... do something with the access port
}
```

6.1.4.3. Working with Tiny Serial

TinySerial is a link layer interface to the KNX bus via Twisted Pair and manages the connection to the KNX Tiny Serial Interface 810 from Weinzierl Engineering. This is a modified TP-UART Protocol without real-time requirements to the host. There is no enumeration provided for TinySerial, and you have to specifiy the serial device (i.e. COM1 for Windows or /dev/ttyS0 for Linux for example) when opening the access port. The default individual address for Tiny Serial is 0x05FF (0.5.255). You can set the individual address using local device management (kdrive_ap_set_ind_addr).

6.1.4.4. Working with FT1.2

This is similar to TinySerial. You have to know the serial device.

6.1.4.5. Sending KNX Telegrams

It is possible to send a KNX Telegram with the function **`kdrive_ap_send`**. All Telegrams are expected in the Common EMI format (EMIC) regardless of the protocol implemented by the Interface Device (the Telegrams get auto-converted in the kdrive C++ library if the Interface Device implements EMI1). EMI2 is not supported.

Example 6.4. Send

The following code sample sends a KNX Telegram (assumes the connection has already been opened):

```
/* Create a KNX Telegram in CEMI format */
uint8_t telegram[] = { 0x11, 0x00, 0xBC, 0xD0, 0x00, 0x00, 0x09, 0x01, 0x01, 0x00, 0x81 };

/* Send the telegram on the Access Port  */
if (kdrive_ap_send(ap, telegram, sizeof(telegram) / sizeof(telegram[0]) !=
 KDRIVE_ERROR_NONE)
{
 ... perform error handling here
}
```

6.1.4.6. Receiving KNX Telegrams

There are two possible mechanisms for receiving KNX Telegrams. Via a blocking receive function and via a callback mechanism. The blocking receive function uses an internal message queue to buffer telegrams as they are received. There is only one receive queue per access port. By default the receive queue is disabled and must be enabled before calling receive. If you only use the receive function to receive telegrams you can simply enable the queue once you have opened the access port. If however you only need to receive one telegram you should disable the queue afterwards otherwise it will continue to buffer telegrams. There is currently no function to clear the queue in the c language bindings. The callback function is in the context of the notification thread (and not in the main processing thread) and subsequently all multithreaded issues (data synchronisation etc) apply. This is especially important if you want to update a GUI from within the callback function (i.e. this generally isn't a good idea as you typically have to be within the GUI thread context to do this).

Example 6.5. Receive (blocking)

The following code sample waits for a KNX Telegram. The timeout is set to 1 second.

```
/* max telegram buffer size */
#define MAX_BUFFER_SIZE (64)


/* telegram timeout: 1 second */
#define TELEGRAM_TIMEOUT (1000)


uint8_t telegram_buffer[MAX_BUFFER_SIZE];
uint32_t telegram_len = 0;

if (kdrive_ap_enable_queue(ap, 1) == KDRIVE_ERROR_NONE)
{
 while (loop_condition)
 {
```

```
  telegram_len = kdrive_ap_receive(ap, telegram_buffer, MAX_BUFFER_SIZE,
 TELEGRAM_TIMEOUT);

  if (telegram_len > 0)
  {
   .. do something with telegram(telegram_buffer, telegram_len)
  }
 }
}
```

Example 6.6. Receive (callback)

The following code sample installs a callback handler for KNX Telegrams.

```
void onTelegramCallback(const uint8_t* telegram, uint32_t telegram_len)
{
 /* do something with the telegram */
}

void mainLoop()
{
 uint32_t key = 0;

 /* Register the callback */
 if (kdrive_ap_register_telegram_callback(ap, &onTelegramCallback, &key) !=
 KDRIVE_ERROR_NONE)
 {
  ... error handling
 }

 while (...)
 {
  /*

     do something else, or block somehow, during which time onTelegramCallback

     will be called

  */
 }

 /* De-register the callback (stops the telegram notifications */
 if (kdrive_ap_remove_telegram_callback(ap, key) != KDRIVE_ERROR_NONE)
 {
  ... error handling
 }
}
```

6.1.4.7. Working with Group Value Telegrams

The *kdriveExpress* API defines a number of functions for working with GroupValue Telegrams, including both Sending (encoding/generating) and Receiving (decoding/parsing). The library supports sending of all Group Value Telegrams, namely GroupValue_Write, GroupValue_Read and GroupValue_Response. The handling of the interactions between the Group Value Telegrams (such as sending a GroupValue_Response in response to receiving a GroupValue_Read Telegram) is implemented in the function **`kdrive_ap_read_group_object`**.

Please Note

The **kdrive_ap_group_write** and **kdrive_ap_group_response** functions require the data buffer length in number of bits and not the number of bytes. This is because it is possible to send datapoint values less than 1 byte (for example, 2 bits) where the data gets compressed into the APCI field.

Example 6.7. Sending a GroupValue_Write Telegram

The following code sample shows how to send a GroupValue_Write telegram

```
/* KNX Group Address, in hex format */
uint16_t address1 = 0x901;
uint16_t address2 = 0x903;

/* KNX Group Value 1 bit boolean data */
uint8_t oneBitBool = 0x01;
if (kdrive_ap_group_write(ap, address1, &oneBitBool, 1) != KDRIVE_ERROR_NONE)
{
 ... handle error
}

/* KNX Group Value 2 Octect Unsigned */
uint8_t twoOctetUnsigned[] = { 0x00, 0x02 };
if (kdrive_ap_group_write(ap, address2, twoOctetUnsigned, 16) != KDRIVE_ERROR_NONE)
{
 ... handle error
}
```

Example 6.8. Receiving a GroupValue_Write Telegram

Once you have a KNX Telegram you can test to see if it is a GroupValue telegram (i.e. GroupValue_Write) and extract its Group Value Data.

```
void onTelegramCallback(const uint8_t* telegram, uint32_t telegram_len)
{
 static uint8_t data[KDRIVE_MAX_GROUP_VALUE_LEN];
 uint32_t data_len = KDRIVE_MAX_GROUP_VALUE_LEN;
 uint16_t address = 0;

 if (kdrive_ap_is_group_write(telegram, telegram_len) &&
     (kdrive_ap_get_dest(telegram, telegram_len, &address) == KDRIVE_ERROR_NONE) &&
     (kdrive_ap_get_group_data(telegram, telegram_len, data, &data_len) ==
 KDRIVE_ERROR_NONE))
 {
  ... on group value(address, data, data_len);
 }
}
```

6.1.5. Services

Management services are required by applications that need to implement some form of KNX device management, for example, an Easy Controller, or KNX device production test software. These services are not required as part of the runtime group communciation. Not all services are enabled as part of the public *kdriveExpress* release. For more information which services are enabled see the api documenation (doxygen). Following services are available as part of *kdriveExpress*: [2]

- Device Descriptor Type 0 Read (Device)
- Device Descriptor Type 2 Read (Device)
- Restart Device Type 0 (Device)
- Restart Device Type 1 (Device)
- Property Description Read (Device)
- Property Value Read (Device)
- Property Value Write (Device)
- Function Property Command (Device)
- Function Property State Read (Device)
- Memory Read (Device)
- Memory Write (Device)
- User Memory Read (Device)
- User Memory Write (Device)
- Authorize Request (Device)
- Key Write (Device)
- Load State Read (Device)
- Load State Write (Device)
- Switch Prog Mode (Device)
- Read Prog Mode (Device)
- Individual Address Prog Mode Read (Broadcast)
- Individual Address Prog Mode Write (Broadcast)
- Individual Address Serial Number Read (Broadcast)
- Individual Address Serial Number Write (Broadcast)
- Network Parameter Write (Broadcast)
- Network Parameter Write (System Broadcast)
- Domain Address Prog Mode Read (System Broadcast)
- Domain Address Prog Mode Write (System Broadcast)
- Domain Address Serial Number Read (System Broadcast)
- Domain Address Serial Number Write (System Broadcast)

For more details information on the services see the kdrive User Guide which contains detailed information for the services, including the input and output parameters and the state machines.

> Example 6.9. Sample services: PropertyValueRead
>
> The following code sample shows how to reads KNX property values connection-oriented.

```
uint8_t data[SERIAL_NUMBER_LENGTH];
uint32_t data_length = SERIAL_NUMBER_LENGTH;
```

[2] the kdrive C++ library implements the complete list of Application Layer Services.

```
/*

 Set the device services to connection-oriented or connection-less

 1 to use connection-oriented

 0 for connection-less

*/
kdrive_sp_set_co(sp, 1);

/*!

 Reads the property value from PID_SERIAL_NUMBER

 of device with individual address 1.1.1

*/
if (kdrive_sp_prop_value_read(sp, 0x1101, 0, 11, 1, 1, data, &data_length) !=
 KDRIVE_ERROR_NONE)
{
 ... perform error handling here
}
```

6.1.6. Easy Push Button Services

The Easy Push Button services are implemented in the KNX-RF interface from Weinzierl. The interface acts as a KNX-RF device and can link with other devices based on a fixed internal channel configuration. Once a link is established the runtime communication for the link can be observed by receiving an asynchronous callback event (the datapoint value indication) and the corresponding communication object value can be read and written. The Easy Push Button Services are only available as part of the commercial SDK.

6.1.6.1. Link Procedure (sensor channel)

To link with a remote device the user application has to know whether the interface is to act as a sensor or an actuator and has to choose a corresponding and appropriate channel from the interface configuration. Note: currently only sensor channels are supported, that is, the interface only defines channels where it can link with remote devices that are actuators. The channel is identified by a channel id. To link with an actuator you have to first put the actuator into link mode (i.e. long press of its link button). Once the actuator is in link mode you can complete the link process by calling **`kdrive_epb_link_channel`**. This function requires the channel id and returns the serial number of the link partner (in this case, the actuator). To remove the link with an actuator (with the interface as a sensor) you have to call **`kdrive_epb_link_channel`** again, and not **`kdrive_epb_unlink_channel`**. This can be somewhat confusing at first glance. To unlink with an actuator simply follow the link proceedure again: place the actuator in link mode and call **`kdrive_epb_link_channel`** with the corresponding channel id.

6.1.6.2. Link Procedure (sensor channel)

The link proceedure when acting as an actuator (i.e. to link with a sensor) is slightly different to the proceedure above, and is currently not supported with the current interface configuration. To link, first place the interface into link mode, by calling **`kdrive_epb_link_channel`** and then press the link button on the sensor. The call to link channel takes a timeout and this should be long enough to complete the link proceedure, i.e. to find and press the button on the remote device. Note however that if the timeout elapses this only indicates that the button was not pressed in time however the interface

will still be in link mode. The link response can still be received asynchronously by the epb callback indication, see also *Section 6.1.6.3, "Callback Indications"*

6.1.6.3. Callback Indications

The Easy Push Button Services implement a protocol that consists of a number of request/response messages and an asynchronous datapoint value indication message. To receive the epb messages that are sent by the interface you can register a callback with the function **`kdrive_epb_register_indication_callback`**. This callback will return all epb messages received from the interface, including the datapoint value indications, and the responses to the request/response services. If you are only interested in the datapoint value indications you can use the function **`kdrive_epb_get_datapoint_value_from_ind`** to both determine if the message is a datapoint value indication and to obtain the datapoint value. To decode the other message responses please refer to the protocol documentation for the WIMAC services. Once you register the callback you will receive a unique key. This key can be used with the standard kdrive access function **`kdrive_ap_remove_telegram_callback`** to remove the callback.

6.1.7. Managed C++ Wrapper

A managed C++ wrapper is provided to integrate the *kdriveExpress* library into .NET. The wrapper simply wraps the C Interface as functions of a C++ class and exposes the Error Codes as a kdrive Exception. Simply add the relevant DLL (depending on the Visual Studio and .NET version) as a reference to your project. Use the object browser to view the API.

kdriveExpressCom

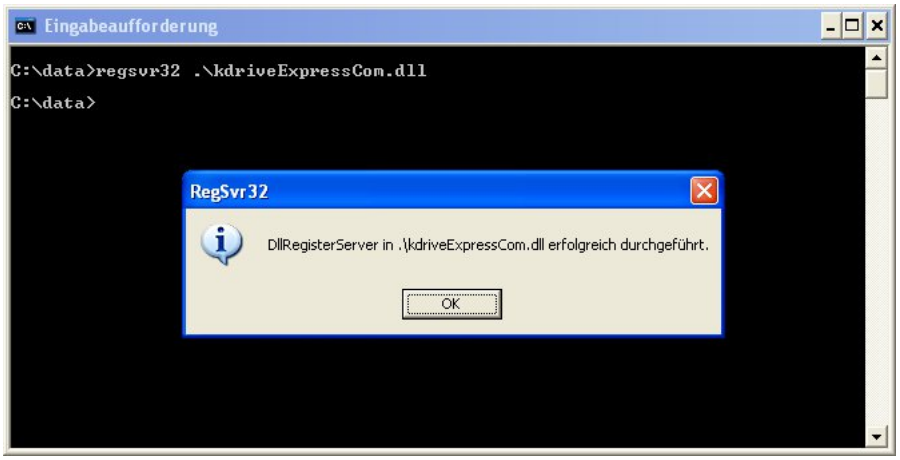
The *kdriveExpressCom* Library is a C++ ATL DLL (Active-X / COM DLL) and provides similar functionality to the *kdriveExpress* however the API is slightly different due to the architecture imposed by the interworking framework. This library is targeted directly towards Visual Basic 6 however could be used with all COM enabled applications.

7.1. Getting Started with kdriveExpressCom

The *kdriveExpressCom* library is written directly to interwork with Visual Basic 6 applications via the Component Object Model from Microsoft. This section assumes that you are using VB6 as your development environment.

7.1.1. DLL Registration

To use the DLL in VB6 you first have to register it. This can be done with **`regsvr32`**. regsvr32 (Microsoft Register Server) is a command-line utility in Microsoft Windows operating systems for registering and unregistering DLLs and ActiveX controls in the Windows Registry. To register the application open the command prompt and enter: **`regsvr32 .\kdriveExpressCom.dll`** . You should get something like the following (depending on your language) to indicate that the registration was successful.



If the registration failed please check to see that the Visual Studio C++ 2008 SP1 Redistribution Package has been installed. See also *Section A.1, "Microsoft Visual C++ Redistributable Package"*.

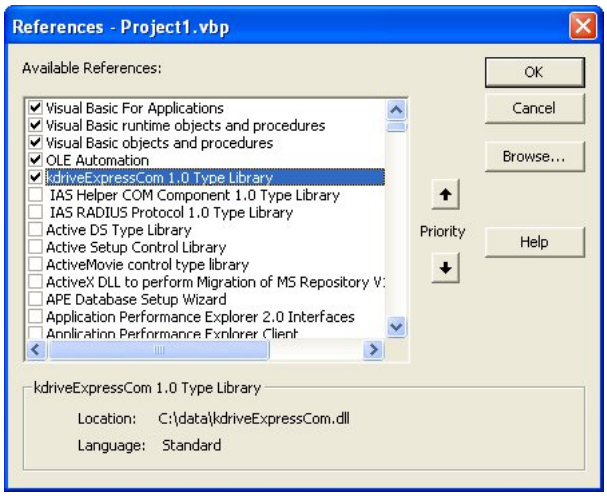
Debugging in Visual Basic 6

We have noticed that the Visual Basic IDE takes a very long time (around 1 minute or so) to exit the application once the kdriveExpressCom DLL has been loaded. After a quick search on the internet we found that other applications also have this problem and ... we have unfortunatly not yet found a solution. If you have any ideas about this problem we would really appreciate your input!

7.1.2. VB6 Add Reference to Project

Once you have registered the DLL you can add it to your VB6 project as a reference. To do this, select Project > References from the menu bar and scroll down to the kdriveExpressCom x.x Type Library (where x.x is the version number). Enable the checkbox to enable a reference to the kdriveEx-

pressCom library and click OK. You should have something similar to the following when you re-select Project > References:



7.1.3. Error Handling

All error handling currently uses AtlReportError with E_FAIL. A description string has been set however it doesn't seem to be translated on the VB side. This issue will be fixed in a subsequent release. The kdriveExpress error codes may also be mapped into user defined error codes within the COM DLL.

7.1.4. API Convention

Standard parameters. All standard parameters types are defined as long. *Data arrays.* All arrays are byte arrays (all arrays must be 1 dimensional, where each element is 1 byte in size).

Appendix A. FAQ

A.1. Microsoft Visual C++ Redistributable Package

If you are using the Windows DLL's you may need to install the Microsoft Visual C++ Redistributable Package (for the specific compiler version, i.e. Microsoft Visual C++ 2008 Redistributable Package SP1 (x86) if you are using the DLL's compiled with Visual Studio 2008 SP1 and so on). Typically if the target machine does not have a copy of Visual Studio you will have to install the redistributable package. They are freely available from Microsoft.

Appendix B. Revision History

Revision 0-0 **Tue Apr 26 2011** **Jason Richards**

Initial creation of book

Revision 0-1 **Thu May 5 2011** **Jason Richards**

Moved Installation Guide into separate article

Revision 0-2 **Wed May 11 2011** **Jason Richards**

Added additional information for Access Ports and working with Group Value Telegrams

Revision 0-3 **Mon May 23 2011** **Jason Richards**

Added Object Server

Revision 0-4 **Fri Jun 03 2011** **Jason Richards**

Added Service Exceptions

Index

K

kdriveAccess, 2, 5
kdriveEasy, 2
kdriveKernel, 2, 83
kdriveObjectServer, 3, 72
kdriveServices, 2, 14